1994

The design and implementation of a flexible object oriented data model for advanced database applications

Gautam Kumar Varma
University of Wollongong

The Design and Implementation of a Flexible Object Oriented Data Model for Advanced Database Applications

A thesis submitted in partial fulfilment of the
requirements for the award of the degree
Master of Science (Honours)
(Computing Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Gautam Kumar Varma, BE, Grad. Dip. Comp. Sci.

Department of Computer Science
October 31, 1994

Abstract

There has been increasing demands placed on the database technology, in the recent past, for providing support for advanced database applications. These applications are characterised by a large amount of data, varying drastically in data type and format with frequent modifications/alterations to the database schema. One of the key requirements of such applications is the need for flexible data modelling facilities and mechanisms for support of flexible objects.

In this thesis, we have proposed a flexible object oriented data model for such applications. The model provides flexibility in conceptual modelling of such applications. A prototype of the model proposed has been implemented under a commercial ODBMS (ObjectStore). A programmatic interface to the facilities provided in the model has also been implemented.

Acknowledgments

For his guidance in research and in the preparation of this thesis, I would like to thank my supervisor Dr. Janusz Getta, for his constant support and encouragement throughout the course of this research.

Table of Contents

Chapter 1 Overview

Chapter 2 Object Databases

Chapter 3 A Flexible Object Oriented Data Model

Chapter 4 Implementation

Chapter 5 Conclusions

List of Figures

Chapter 1

Overview

Apart from traditional data processing applications, there have been increasing demands placed on the database technology, in the recent past, to provide solutions for advanced database applications. The relational data model lacks facilities related to transaction mechanisms, support for complex object and data modelling facilities to accommodate such applications gracefully. The objective of this thesis is to develop a flexible object oriented data model for advanced database applications. In this chapter we present the characteristics of these applications, and reason as to what makes this new application domain so difficult to accommodate in the traditional relational model. Finally, we outline the contents of the thesis.

1.1 Application Domain

The advanced database applications include CAD/CAM (Computer Aided Design and Manufacture), VLSI (Very Large Scale Integration) [Gupta91] databases, CASE (Computer Aided Software Engineering) [Brown91], Ill-structured information systems [Kotteman90] and Office information systems [Ahsen90] [Lochovsky90]. Others include application domains dealing with spatial [Woelk90] and image databases [Brolio89] which require flexible modelling mechanisms and efficient data representation mechanisms. Emerging needs from application domains include support for new and evolving technologies such as active databases with support for rules and triggers [Gehani92].

The advanced database applications may be characterised with a large quantity of complex, shared and concurrently accessed data to be managed. Other requirements

enforced by these applications relate to reliability, distribution of data storage and processing over networks, design orientation, e.g. computer-aided design of complex artefacts such as circuit design, manufactured goods and software. One key requirements of advanced database applications relates to the need of flexibility in conceptual modelling and extensibility.

The application of database technology to these application domains is an extremely important and emerging research area. Many of these applications deal with a large amount of objects that are composed of other objects. For example, a part in the part hierarchy may be composed of other parts, an integrated circuit may be composed of other modules, pins and wires. A complex geographical information system may consist of a variety of data formats including graphics, text, sound etc. A program module may consist of other program modules, each with a declaration part and a body. A document may be composed of sections and the sections themselves may be composed of section headings, paragraphs of text, and figures.

Furthermore, these applications need to perform large and varied amount of processing on the application data stored in the database. In such applications, complex objects are the units of processing and operations on composite structures. For instance in a design application, it may be necessary to lock an entire assembly component or delete a part (which is itself composed of other parts). In such a case the deletion must be propagated to all sub components of the part.

1.2 Characteristics of Advanced Applications

The advanced database applications are strongly characterised by schema and data evolution. Unlike relational systems, changes to the schema are expected and are quite

frequent in nature. Many users will need to change the schema to include new data types into the system as the system evolves. Furthermore, data stored in the database is primarily structured in nature.

The stored data includes a variety of diverse data types including graphical data, design documents and programs. Also, data items may be large, complex, and of variable length. The database typically includes many entity types. Complex relationships may exist between entity types. Additionally, new relationships may be created with the entity types which are already stored in the database. The database is characterised with a strong evolution of database structure and contents.

These applications require support for complex objects and transaction mechanisms which extend beyond the traditional mechanisms. The traditional transaction mechanisms, based on record locking, do not scale well for long duration transaction - an important characteristic of advanced database applications. Furthermore, there is a need to support advanced technologies such as deductive databases, active databases, triggers [Gehani92] and alerts. Another issue of importance relates to flexibility in support for modelling constructs.

We can gain some insight into the differing requirements of a database system for advanced database applications by considering characteristics of a typical Computer Aided Software Engineering database interaction [Brown91]. For example, when a software engineer is required to make an update to a program module, the following actions may take place. Firstly, having located the original program module within the database system, some way of obtaining exclusive access to that module is required. To perform a change to the module, the software engineer may require access to a large number of other data items. These data items may include the original specification of the program

module, the bug report that necessitated this change, related modules which provide input to this faulty module, details of the previous test and so on.

Depending upon the changes required, it may take anything up to a number of weeks to complete the amendments, and obtain approval from the quality assurance engineers that modules can be released for use. A traditional database system does not provide adequate support for complex objects. Furthermore, it does not support long duration transactions, which are critical to support such an application.

Exclusive access to large amount of data in the database would require data to be made inaccessible to others through the database locking mechanism. In many cases, this prevents such data even from being read by other users for the whole database interaction which may last for hours or even days. Additionally, the maintenance of complex relationships among a variety of entity types is difficult to support in a traditional database system. Even with this simplified example, we can have some appreciation of the complexity of relationships which commonly exist in software engineering databases.

Another issue of grave importance is the notion of a version relationship between data items. While simplified version mechanisms can be modelled in a traditional database system, the importance of versions and configuration to software development applications leads to a number of ramifications with regard to database storage and performance. Also, there are complex integrity constraints that must be enforced. It is difficult to envision how the traditional database constraint mechanisms can be extended to provide support to such kind of requirements.

From the above discussion about the characteristics of advanced database characteristics, we can identify flexibility and extensibility as two of the most important

requirements of such applications. Flexibility in the support for complex objects and operations which are performed upon them, is highly desirable in such design environments. Additionally, extensibility of the type system to be able to integrate new data types (application specific) may also be seen as highly desirable in such environments.

1.3 Why not Relational Data Model

The relational systems have been around for about two decades and much work has gone into optimisation of their query languages, retrieval and storage structures. Yet for advanced database applications, it has been found that the relational systems are inadequate for a variety of reasons. So, what is wrong with relational [Date90] systems? Why they cannot handle these advanced database applications?

The traditional data processing applications are characterised by a need to manage a large amount of data which is rather static in nature. Furthermore, these applications do not require support for a large amount and diverse data types. A major component of database research in the last decade has been focused on improving the efficiency of these database characteristics. Such improvement includes making the query language more efficient and providing support for application programming languages. This has lead to the development of techniques which make databases highly efficient in commercial environments, but inadequate for application domains whose characteristics are different from those of commercial applications. In situations where commercial database systems have been applied to advanced database applications, it has been found that mechanisms and services of relational databases systems are not particularly appropriate for advanced database applications.

We can identify the characteristics of advanced database applications which set them apart from those of the traditional database applications. Most information in a commercial database system is static in nature and can be described a priori so that the schema is static and pre-compiled. Furthermore, updates to the schema are infrequent and are tightly controlled by a group of database administrators. The data stored is atomic in nature and is typically of fixed length. The data types of stored information are typically small in number, with large number of instances of each type. Mostly, only simple relationships exist between various entity types. Further the database is initially loaded with a large amount of data and no evolution of data schema. The data items which are updated in the database, are typically single valued and are often updated in place. Lastly, the transactions of the applications are typically short, atomic and can be used for concurrent access to the data.

1.3.1 Performance Issues

The bottle-neck in relational systems arises from the performance issues in the retrieval of data from database [Maier89]. While queries for associated retrieval of attributes from the disk are quite adequate from the perspective of performance, the grey area in the performance of relational based systems is in accommodating design and modelling applications.

Such applications also require a strong support for complex objects. Representation of such complex objects in a flat relational structure leads to performance problems. The problem arises as a consequence of the first normal form constraint which requires that the object space must be mapped onto a collection of flat relations. This results in the inherent semantics of objects to be lost because of composition. Performance problems also arise in accessing a complex object as a composite entity.

Such application are also characterised with complex objects being a single unit which are used for the access, computation, storage, locking, and physical placement. Since each complex object must be constructed from composite entities which are stored in other relations, the simple task of constructing a complex entity from it's components is an expensive task. The task involves a large amount of processing in order to build up the complex entity from tables in which it has been represented.

Complex information systems typically demand high performance. For example, in order to test a circuit in design applications, the designer may need to pass a variety of test cases to check the range of the output. In some cases, computation on the output results may also be needed. In CASE we need to efficiently load modules from the database into memory and have the ability to search for keywords in an efficient manner. Information Systems and Office Information Systems need the ability to quickly search for some information in a vast data space.

1.3.2 Data Modelling Issues

By it's very nature, a database system is a collection of application domain entities which have been modelled from the problem domain entities. The most fundamental principle on which modelling rests is a one-to-one correspondence between proxy objects in the database and the entity objects in the real world, the proxies represent. In doing so, a database tries to faithfully model the real world entities, thereby creating the illusion that one is dealing with the real world entities [Kent91].

Naming is important to make modelling work as it provides a unique identification of application domain entities. The one-to-one mapping between the real world entities and the application entities is typically maintained with various unique keys with values of

entities spread over many relations. These unique keys include primary keys, foreign keys and candidate keys.

The problem with this approach rests on the very assumption of making a mapping between real world entities and application domain entities. Using unique identifiers to model the real world entities, a number of problems are introduced. The real world entity which we are trying to mimic in application domain may not have a unique identifier or worse, may have many unique identifiers.

Other problems arise due to the lack of adequate data modelling facilities found in relational database system. The user is often limited to types which are made available with the database system. New types, not pre-defined in the system, are difficult to create. Consequently, the application developer is limited to the types which are made available in the system. Making a new entity which is composed of other entities results in an application entity, the value of which is spread over a number of relations. Every time such a composed entity is desired to be accessed, the entity must be constructed from it's constituents relations. This has disadvantages in corrupting semantics of the composed entity. Performance problems result as, each time the composed entity needs to be accessed it involves a penalty of composing objects from attributes which are spread over in other tables.

Additionally, the value based semantics of relational databases do not allow the application designer to model the real world entities in an adequate fashion. Constraints arise from limited data modelling constructs and inability to make new types from pre-defined types. The advanced applications also need mechanisms to provide support for multimedia objects, which may constitute of sound, graphics and arbitrary large amount of

text. The relational model provides no built in constructs for the support of such data types with efficient representation mechanisms.

The presence of two languages one for data manipulation and the other host programming language (into which Data Manipulation Language commands are embedded) presents the infamous impedance mismatch problem. This results in much complexity of applications, as there are two data models to be dealt with; one relating to the programming language and others relating to database. Types provided in the two data models are quite different. The programmer is thus required to bridge the gap between the two models. Such an environment is a rich source of application complexity and a frequent cause of errors.

The above said factors may not be important for the case of banking transactions, inventory management and the like, as the relational systems were partially developed and optimised for such applications. However, relational systems have proved to be a severe bottle-neck in case of Engineering Information Systems. This has lead to a number of research efforts in the past for developing new frameworks, application specific environments and customised data models for such applications.

1.4 Transaction Processing

In addition to above stated criterion, there are additional requirements for transaction processing needs of advanced database applications. The transaction model found in traditional databases is found lacking severely in terms of functionality and efficiency when used for these new applications. Efficiency is particularly important considering the high throughput demands placed on these complex information systems. In terms of functionality, traditional transaction models were assumed to be short lived

and were targeted for competitive applications. Such applications are characterised by many users attempting to access a piece of data concurrently for a short duration of time.

Activities in complex information systems tend to access many objects, involve lengthy computations and are interactive in nature, i.e. pause for user's input. Even in those cases where activities with such characteristics can be modelled in the traditional transaction mechanism, they degrade the system performance to unacceptable levels. Moreover, endless and collaborative activities which are typically found in these systems cannot be captured by traditional transaction models due to serialisability as the correctness criterion [Korth90].

The need to capture reactive (endless), open-ended (long-lived) and collaborative (interactive) activities found in new applications suggest requirements for co-operative transaction processing models. In order to fill the transaction processing needs for these complex information systems, variety of transaction processing models have been proposed. These include the Nested Transaction Processing Model, Recoverable Communicating Agents, Co-operative Transactions, Split Transactions and transaction groups [Chrysanthis90].

1.5 Approaches

The research in advanced database applications has yielded two categories of support environments. These are application specific environments and environments built using the traditional database management systems. In this section, we briefly review the approaches that are prevalent for accommodating the advanced database applications. There are two possibilities for dealing with the inadequacies of relational database systems.

In the first approach, a commercial database system is used with a further layer of software providing additional database services. For example, the existing locking mechanisms may be replaced by new transaction mechanisms which support extended periods of database interaction. In addition, support for multiple versions of database objects may be added. While this approach avoids the need to write many database facilities from scratch, it can still involve a significant amount of work. Furthermore, many of added features are "excess baggage" and may severely impair the overall performance of the enhanced system. With a majority of the database systems, it is not possible to take advantage of some of the services provided, while overriding others. Not only the systems are closed to such attempts, but it may also take a great deal of effort to add new services in a seamless and integrated fashion.

The second approach is characterised by developing [Premeriani90] [Brolio89] [Kotteman91] a new set of data management capabilities, instead of using a database management system. These capabilities are then tuned to the requirements of a support environment. This approach in the light of existing commercial database systems represent many years of production effort. It is a major undertaking to design and implement a new database system. Such a system would have the advantage that services provided would closely match with requirements of the application domain.

In the recent past, there have been a number of major research efforts directed at the problems of database support for advanced database applications. Not only the requirements and design characteristics of engineering databases been debated, but also attempts have been made at designing and implementing general database systems for such applications domains. A variety of solutions have been the outcome of these projects. While there has been a number of issues which have been and still are subject of much

debates, it has been generally accepted that ODBMS seems to be providing the most promising way forward.

1.6 Conclusions

In this chapter we have presented an introduction to advanced database applications. We also reviewed characteristics of advanced database applications and reasoned as to why the relational model cannot accommodate these applications. Issues relating to performance, data modelling and transaction processing were identified as some of the key areas where relational model was found lacking in the context of advanced database applications. Another important characteristics of advanced database applications is the need for flexibility. The need for flexibility in modelling database applications was clearly identified. Object orientation was also identified as an attractive environment under which to provide such a flexibility.

We are aiming at developing a flexible object oriented data model for advanced database applications. Instead of developing the desired capabilities of such a system from scratch, we have used a commercial database system in implementation. Such a system provides much of the functionality we desire in our model.

In Chapter 2, we firstly present a survey of data models and architectures of the next generation database systems. We, then survey the next generation database systems for advanced database applications. These database systems which allow higher performance and superior data modelling capabilities are presented. The capabilities of these systems and various strategies in implementing them are also proposed.

In Chapter 3, we propose a data model for these advanced database applications. The various aspects of the model, i.e. flexibility and extensibility are emphasised. Chapter 4 presents the design and details of implementation of the proposed extensible data model. It also describes in detail the implementation and various features of the system. Finally in Chapter 5 we, discuss conclusions and avenues for further research.

Chapter 2

Object Databases

Object database systems are the focus of current research and development efforts. A number of research prototypes have been implemented and commercial systems have just made their appearances in the market place [Lamb91] [Butterworth91] [Arango91]. Despite this activity, there seems to be little consensus among researchers in the field as to what is an ODBMS. Disagreement even on the desirable features/characteristics [Participants89] of ODBMS are not uncommon.

The architectures upon which ODBMS have been implemented are equally controversial. A variety of approaches have been adopted with mixed results. No consensus has been achieved on the preferred approach to the construction of an ODBMS. In this chapter, we focus on some of the extensible database architectures. These architectures have been discussed with regards to the extent of flexibility that they provide. After the presentation of database architectures, we discuss some of the desirable characteristics of an ODBMS.

2.1 Database Architectures

In this section, we take a look at various database architectures. We discuss some of the interesting database architectures. Distinct from the traditional monolithic approaches to implementing databases, database system generators and object managers present another perspective to database management systems. These database architectures are discussed in view of features of flexibility and extensibility that they

provide. We now present a description on database programming languages, database system generators and object managers.

2.1.1 Database Programming Languages

Database programming languages represent database systems that extend existing programming languages to incorporate persistence, concurrency control and other database capabilities. In database programming languages, both query language and programming language execute in the same environment, sharing the same type system and data workspace. This architecture provides a single environment for database objects, which appear semi transparently as programming language objects.

Many ODBMS based on the object oriented data model may be referred to as object oriented database programming languages. These systems are based on the database programming language architecture. In such an architecture, applications are written in an extension of an existing programming language. The language and it's implementation (compiler, pre-processor and the execution environment) have been typically extended to incorporate database functionality.

Most object oriented database programming languages have been incorporated with C++ [Stroustrup91] or a derivative thereof [Schuh]. A few implementations have focused on Smalltalk [Butterworth91] or a LISP derivative [Fishman90]. Many of these database programming languages also provide a procedure call interface with C. However, the interface is not a complete integration, in the sense that C data structures cannot be made persistent transparently. Some proposed prototypes have been able to integrate successfully with more than one language such as GemStone with Smalltalk [Goldberg89] and O2 [Arango91] with LISP and a C++ derivative.

The distinction between a database programming language and an extended database system is actually more a continuum than a two way categorisation because there are degrees of integration between the programming language and the database system. COP [Andrews90] may be regarded as an object oriented database extension of C, or as a new language altogether. Nevertheless, current object oriented database system are biased towards database programming languages. Additionally, almost any of the object oriented OODBMS's can be transformed into an object oriented programming language model, as the object oriented data model and the programming language models are similar.

2.1.2 Database System Generators

Database system generators represent a system which allows a database implementor to construct a DBMS tailored to the needs of a specific application domain. Database system generators do not enforce their own data models. They allow a DBMS designer to tailor the data model, type system, query language and optimiser for a specific application domain.

Database system generators typically include reusable libraries which aid a database implementor in the selection of appropriate components and their characteristics to fabricate and automate the generation of a database system. These systems differ in the extent to which the system allows in the generation of customisable DBMS and tools to facilitate automatic or semi automatic construction of a DBMS. For example, the Exodus [Carey90] prototype allows plug compatible components to be integrated. On the other hand, the Genesis prototype only allows few of features of a DBMS for advanced application domain.

The Exodus approach is more flexible; it permits a much wider collection of tools to aid in the construction of application specific DBMS. The data model is defined by a special user called a database implementor. Both the conceptual data model and the internal data model must be described. The conceptual data model specifies the query and the representation capabilities of the system, such as keys, relationships and the query syntax.

The internal data model specifies access methods and ways in which conceptual data schemas can be mapped to them. Additionally, the type system of the database is customisable and extensible in that new data types may be added. Application specific database systems include access methods and operators which are appropriate for their intended domain of applications. These methods are required to be implemented by the implementor to suit the application domain.

These systems provide extensibility and performance by tailoring components of a DBMS to suit an application domain. Database system generators may be seen as another end of the spectrum in database architectures.

2.1.3 Object Managers

Object managers [Cattell91] fill the gap between database management systems on the higher end and file systems as repository of persistent data on the lower end. File systems are unreliable and provide no data consistency. Furthermore, sharing between applications is primitive at best. DBMS support sharing, data integrity and transactions. Moreover, DBMS are scaleable and tend to be robust but are often an order of magnitude slower than file systems.

Object managers provide the ability to persistently store simple objects, and generally provide multiusers concurrency control. However, object managers lack a query or a programming language. These systems may be thought of as an extension of virtual memory. We may summarise object managers as a flexible and persistent storage of data. These characteristics facilitate construction of arbitrary model for transaction management, access control, and configuration management.

An object manager generally has a limited data model - at a physical level only. It provides an untyped repository for persistent objects and does not have all features that an ODBMS provides. They may be seen to be providing the minimum basic services which may be used for the construction of a wide variety of related data management software. Object managers, thus permit the development of applications which need to store data persistently without the baggage of other capabilities provided by a typical DBMS.

The Kala basket [Simmel91] mechanism is an example of such an object manager. Kala is an untyped persistent store for object based systems, such as OODBMS, OMS and object oriented languages with persistence. The grouping of data elements is managed by Kala in terms of baskets. Baskets are dynamic grouping of immutable data elements managed by Kala. Furthermore, baskets are also responsible for synthesising transactions, configuration management, access control semantics.

2.2 Databases Characteristics

In this section we identify some of the desirable characteristics of ODB systems (ODBS). It must be pointed out that a number of issues relating to object orientation need to be resolved. This list of characteristics may be seen as desirable though not complete list of characteristics.

2.2.1 Complex Objects

The advanced applications deal with the design and manipulation of complex objects. Objects consist of multiple sub components, which may themselves be complex objects. Numerous approaches to database support for complex objects have been emphasised, as is evident from the large number of research prototypes and papers which have been published in this area [Khoshafian90a] [Dayal90] [Manola90].

The basic problem in the representation of complex object is that in conventional database systems a complex object is typically represented by many tuples scattered among several relations. The tuples in these relations that together constitute a complex object, have to be logically linked by values to compose the complex object. Operations on complex objects necessitate composing the complex object from multiple relations. This leads to severe performance penalties as a single operation leads to the execution of multiple operations. Furthermore, the inherent semantics of complex entities are lost. Extensions to the relational model have been proposed. These extensions concentrate on the ability to treat complex objects as a single unit for querying, locking and physical storage.

Considerable effort has been made in the past work on complex object to overcome this deficiency of relational model. There have been several attempts to address this issue and to introduce some generality into the relational model by relaxing the first normal form constraint. The relational model has been extended to represent hierarchical structures by the notion of nested relations. The relational algebra and calculus have correspondingly been extended to manipulate and retrieve such hierarchically structured objects.

Another aspect of complex object relates to the ability of complex objects to share subparts. Often CAD, CASE and other design environments provide a library of components, with expectation that designs can make use of these components. Object identity [Khoshafian90b], facilitates sharing of complex object's subparts. Each object, in a system supporting the concept of object identity, can be uniquely identified by a system generated identifier. Such an identifier can be used to refer to a shared subpart in multiple complex objects.

2.2.2 Data Types and Classes

The distinction between data types and classes is indeed subtle. This distinction is complicated by two differing views presented by programming language researchers and the database community. Some authors have chosen to overlook the difference and brushed aside the topic referring it to as very complex and the difference adding more confusion than value [Booch91].

However, Data types and classes are two different notions. Some programming languages provides built-in constructs which distinguish classes and types. The type system evolved in programming languages as a means of characterising values that arise dynamically in the course of computation. The type system of a language is the manner in which a set of values over which a computation may range is partitioned into subsets. These subsets are arranged such that only values within a particular type may occur at a particular point in the computation specified.

Depending upon the language, a number of flavours of data types or their facilities may be offered. [Albano89a] describes a useful framework for comparing type systems. A database programming language may offer three kinds of type facilities. Abstract types

permit new types to be defined by specifying a representation type and new type operators to be defined. The representation is hidden in an abstract type. A concrete type definition is an association between a name and a type expression which makes that name completely equivalent to that expression. Thus a concrete data type is not a new type but is a new denotation for the representation mechanism. A language offers unnamed types when a type expression can be used in situations where a type identifier could appear.

The notion of data type also permits type checking to be performed over programs. Type checking is intended to signify a check made on a program to determine if types are defined and/or used in a consistent manner. There are two principal forms of checks: one which concerns only the definition of the schema, and the other which concerns the use of data. Static checking of programs is very helpful in large scale software development as the erroneous use of types are flagged at compile time rather than waiting at the run time.

The term class has been used in the literature to refer to a variety of concepts [Cattelle91]. The first of these concepts refer to the use of classes to define an object type or the intent of an object. Intent defines the structure (i.e. attributes), relationships (in which objects having this type can participate), and behaviours (i.e. methods). There may be multiple implementations of the same external structure and procedure.

A class may also refer to an extent. The extent in this context refers to a set of objects of a type. This is sometimes referred to as a classification of objects, so the extent might appear to be the most appropriate meaning for a class. For example, in Galileo [Albano89b] a class is the mechanism to represent databases by means of sets of modifiable interrelated objects. A class is the collection of constructed objects of an abstract type and is characterised by a name and the type of it's elements. The name of a

class denotes the sequence of elements of the class currently present in the database (class extension), while the type which must be an object type represents the structure of elements (class intention).

Finally, a class may correspond to the notion of an abstract data type consisting of an interface and implementation of the interface. Abstract data types permit new types to be defined by specifying a representation type and new type operators. The representation type is hidden in an abstract type; only the interface is visible to the user of a type. This separation between the interface and implementation permits a type to have multiple implementation. The implementation of a type may also be changed while the interface remains the same.

2.2.3 Encapsulation

Encapsulation is a technique for minimising interdependencies among separately-written modules by defining strict external interfaces. These external interfaces may be seen as a contract between the program module and the client. If the client depends only on the external interface of the module, the module may be re-implemented without affecting any clients.

Encapsulation helps in cleanly distinguishing between the specification and implementation of an operation thereby enhancing modularity. Modularity is necessary to structure complex applications designed and implemented by a team of programmers. Advantages of encapsulation may be maximised by minimising the exposure of implementation details in external interfaces [Snyder90].

There are two views of encapsulation: a programming language view, and a database language view. According to the programming language view, an object has a data part and an interface part. The data part defines the internal data structure of the object, whereas the interface part defines operations which are possible on the object. The database translation of the principle of encapsulation is that an object stored in the database encapsulates data along with methods.

Considering an ODBS, we may store all attributes of the entity Employee in a class. In addition to the data definition of Employee, we also store the operations which are permitted on an instance of class Employee. Each object of class Employee encapsulates data and operations which are permissible on it. This is accomplished by storing data along with the operations in the database.

This is in sharp contrast with relational systems wherein there is no encapsulation of data and methods which operate on the data. Encapsulation is violated every time when an application programmer is burdened with the task of writing an application to increase the salary of an Employee. Such applications are written in an imperative programming language where the programming language is embedded with data manipulation constructs.

2.2.4 Inheritance and Type Hierarchy

Inheritance is an important mechanism for specifying relationships between types. It may also be used to express generalisation. Inheritance is a relationship among types wherein one type shares the structure or behaviour defined in one (single inheritance) or multiple types (multiple inheritance). Furthermore, inheritance is a powerful modelling tool, and it helps in factoring out shared specifications and implementations in

applications. Inheritance allows types/classes to be linked together in a type/class hierarchy.

Generalisation is also referred to as inheritance sub typing or subclassing. There are primarily four kinds of generalisations: Specification, Specialisation, Classification, and Implementation [Cattell91]. This classification may be seen as a consequence of different meanings associated with the term class.

One kind of generalisations commonly associated with the ODB is Specification Subtypes may be defined as type predicates applied to objects as in programming languages. The predicate may have to be applied at run time if the subtype is defined based on a particular attribute value. The second is that of classification wherein subtypes may simply be used as sets to classify objects that is, to define different type extents. For example, we could classify a document as a book, a journal. It should be noted that classification is really just a trivial case of specialisation. Thus, we could define a special attribute of document that specifies to which subtype it belongs.

The third type of generalisation is referred to as Specialisation. Specialisation refers to the ability of subtypes to add additional attributes or methods to a subtype. In specialisation, a subtype may add attributes or methods to a subtype. The final type refers to Implementation which refers to the fact that subtypes may have different implementation of methods defined in the subtype.

Note that the uses of subtypes for these four different purposes are not mutually exclusive. Database systems and programming languages generally do not have different subtype mechanisms for these kinds of generalisations. Subtypes are used in combination with inheritance to provide a convenient abstraction mechanism for all four kinds of

generalisations. A subtype inherits the type intent of it's super type. The intent as discussed previously includes the definition of all of properties associated with a type.

Type hierarchies may have more sophisticated semantics. Types may have multiple super types, or objects may have multiple types. ODBMS provides different mechanisms to define subtypes and classify objects into subtypes. If objects have multiple types or objects types can be defined by predicates, it may be desirable to have additional integrity constraints in the type system. Most of the ODBMS allow a type to have multiple super types. This feature is generally referred to as multiple inheritance. because such a type inherits properties of multiple super types.

2.2.5 Object Identity

A database system may be visualised as a collection of application domain entities which have been modelled from the problem domain. The most fundamental principle on which this modelling rests, is a one-to-one correspondence between proxy objects in a database and entity objects in the real world, the proxies attempt to represent. In doing so, the database tries to faithfully model the real world entities, thereby creating the illusion that one is dealing with the real world entities [Kent91]. While a database does not guarantee realism in the sense of correctness, techniques such as naming, constraints and object identity help maintain plausibility.

Naming is important to make modelling work, as it provides a unique identification of application domain entities. The one-to-one mapping between real world entities and application entities is typically maintained in relational systems by various flavours of unique keys with values of entities spread over relations. This approach leads to many problems. These problems include, the real world entity which we are trying to mimic in

application domain may not have a unique identifier or worse, the real world entity may have many such unique identifiers.

Object identity [Khoshifan90b] is a unique identifier that distinguishes an object from the rest. Such a unique identifier is system generated and is guarantied not to be used for the entire lifetime of the system. This conceptual separation of identity of an object from the name of an object is important. Such an approach does not compromise the concept of identity of an object with its name. Object identity, thus allows us to handle aspects of naming and identity in a much cleaner fashion.

Not only the concept of object identity is important for the case of conceptual aesthetics, but object identity is also important as it serves as a data modelling construct. Object identity permits us to identify whether two objects are identical or equal. Equality of two objects refers to the case wherein the two objects in question have the same attributes. Identical objects refer to the case where two objects have identical values for their attributes. The use of keys for naming and identification of entities in relational systems makes such a distinction impossible.

2.2.6 Extensibility

One of the distinguishing characteristics of advanced database applications relates to the need for extensible. CAD, CAM, CASE and spatial applications need ability to have flexible representation mechanisms for a variety of data types. There must exist mechanisms to provide hooks for extensibility of data types. This relates to the addition of new data types in the type hierarchy.

A variety of approaches have been adopted by researchers in this area. Some have focused on the development of application specific environments [Gupta91] [Brolio89] [Afsarmanesh90] which provide support for extensibility. Others have approached the issue of extensibility by building a complete ODBMS constituting of the functional components and then, providing hooks for extensibility. Such hooks for extension include support for user extensions within the context of a full-functional ODBMS. This approach is being taken for example, in [Rowe90] and [Lohman91] systems. Postgres [Stonebraker91], for example, allows procedural data as a data type.

Yet another interesting approach that has been adopted relates to the toolkit approach of constructing an extensible DBMS. Under this approach, a library of components is provided. These components may be used to automatically/semi-automatically construct a DBMS, tailored for a specific application domain. This approach has been investigated by the Genesis [Batory90] and Exodus [Carey90] research prototype.

The toolkit approach is general in it's ability to construct a customised ODBMS tailored for a specific application domain. However, the other approach of building a complete ODBMS with hooks for extensibility is more general, in that a single ODBMS is able to provide extensibility for an entire domain of applications.

2.2.7 Persistence

The concept of persistence was introduced almost a decade ago. Early work in the area of persistence established that it was an area of language design and was not possible to incorporate it into a language. However, the classic work of Atkinson in the area of

persistence established that it was indeed possible to incorporate the notion of persistence in a programming language as demonstrated with the work on [Cockshott90] PS/Algol.

Persistence refers to the property of a data type which allows any of its instances to exist for an arbitrary time. Two important principles related to persistence are that of persistence independence and persistence data type orthogonality. Persistence independence refers to the independence of how a program manipulates data values. Conversely, a fragment of program is expressed independently of the persistence of the manipulated data.

The principle of persistence data type orthogonality refers to the ability of any data type to be made persistent. In addition to these principle it is also desirable that these features be provided without perturbing aspects of the language design. Persistence is an essential feature of an ODBMS system with various strategies being used to incorporate persistence into a programming language.

A programming language is typically extended with constructs to incorporate persistence. Notable example of this architecture of an ODBMS include ObjectStore [Lamb91], GemStone [Butterworth91] and O2 [Agarwal89]. Another approach that has been investigated to incorporate persistence in a programming language or building an ODBMS to support persistent data types, is based on inheritance [Dixon89].

2.2.8 Secondary Storage Management

In this section, a discussion of various implementation aspects with regard to the ODBMS is presented. Specifically issues related to storage management, storage access, clustering, and impact of inheritance are discussed.

Most ODBMSs are implemented as logical database shells requesting data from underlying storage managers. This approach has the advantage of divorcing low level physical data management and storage details from their higher level interpretations and use. In an ODBMS, the storage manager or object server is responsible for the allocation of disk space to objects, placement of object on the disk and so on. The higher level system interprets users operations within constraints of a particular data model. These operations are converted into request for objects which are issued to the object server.

As compared to the fixed record structure of relational databases, ODBMSs typically have a deeply nested structure, supporting complex object, user defined types and methods which operate on data to be stored along with the data itself. The Encore [Zdonik87] system uses object server which maintains minimal semantics about the stored object. These objects are treated as uninterpreted blocks of storage. On the other hand, the GemStone [Butterworth91] ODBMS records information about the internal structure and some semantics of objects that are stored. The Encore approach provides a neutral storage mechanism capable of being used as basis of many different high level interpreters. In the GemStone approach, the storage manager can make use of the extra information about objects that it has stored in performing disk storage allocation strategies, object retrieval and other capabilities.

Another implementation issue in this context is the identification of objects within both components of the system i.e. the object server and the higher level interpreter. For example, the object server may identify objects using physical disk addresses, while the higher level interpreter may use its own logical object identifiers. If two distinct object references are maintained, then the mapping between them must be maintained and object de referencing must take place when an object is moved from/to the server. This can degrade the performance of the system to unacceptable limits. If internal physical address

are used in both components i.e. the object server and the higher level interpreter, physical data independence is lost.

There are two primary methods for storing objects: relation based storage format and object based storage format. In the relation based storage mechanism, objects are decomposed into constituent fields. Each constituent field is a binary relation containing an object identifier and a value of the field. Object based storage mechanism groups all fields of each object together on to the disk. Additional storage issues must be considered when considering the storage of objects which belong to a class hierarchy.

There are three approaches to the storage of object hierarchy: Naive approach, No Duplication approach, and Splitted Object approach. In the Naive approach, a direct implementation of the logical view is implemented, where as in the No Duplication approach, the objects are stored at the lowest possible level in the class hierarchy. In the Splitted Object approach, each instance of a class is stored together with the value of all properties which are characteristic of that class.

Furthermore, problems with excessive disk access is exacerbated when highly structured data items are represented in a data model with a single set of structuring primitives. For example, in the relational model complex nature of a design document could be modelled explicitly by defining a number of relations representing the document, document section, and paragraphs of text within those sections. Operations on documents will require access to all of those relations, involving significant performance overheads with conventional database clustering techniques.

Computer aided design applications also require significantly large amount of data to be recorded. In fact it is possible that a single object may be very large, perhaps larger

than the main memory of computer system. Thus, techniques must be implemented for dividing large objects in a controlled fashion, allowing individual pieces of an object to be examined and used. The EXODUS [Carey90] system has addressed these issues by implementing a novel storage algorithm which essentially allows arbitrary large objects to be stored without the significant overhead in reading writing components of a large object.

2.2.9 Concurrency and Transaction Support

Whenever concurrent multiple access to shared data is possible, some concurrency control mechanisms must be implemented to prevent corruption of data by interference. In commercial relational databases, concurrency control is commonly implemented by locking mechanisms. There are three types of locking techniques: read-locks, write-locks and read-write locks.

However, this simple approach of locking to provide concurrency is of limited use for design application which need to access data for long duration. It is not possible to enable the application to lock a large portion of the database for a long time. Other mechanisms that allow uses of database by multiple designers for extended duration of time, need to be explored.

One approach that has been exploited is aimed at a more optimistic approach to concurrency control. If we assume that it is rare for two users to amend the same set of objects concurrently, then it is possible to relax the usual approach to locking of data objects. Much concurrency is possible for applications, when including the notion of multiple object versions. In this case, access to a particular object is specified by a version number. This allows multiple users to access multiple versions of the object without conflict. Changes to an object no longer cause conflicts, but create a new version of an

object. This allows users to make changes to a new version of the object. However, possibility of conflict still exists when multiple users are trying to access the same object version.

The transaction model found in traditional database system is found lacking in terms of functionality and efficiency when considered for new and complex applications. In terms of functionality, traditional transactions were assumed to be short lived and were targeted for competitive environments. This emphasises processing large number of short simple transactions issued on behalf of users who are oblivious of each other.

Activities in complex information systems tend to access many objects, involve lengthy computations, and are typically interactive. Such activities cannot be gracefully accommodated within the traditional transaction model. Even for the case when such activities can be modelled in classical relational model, they degrade the system performance to unacceptable levels.

Design applications are characterised by groups of co-operating users who need parts of the database for long period of time. Transactions in context of business data processing may be seen as a unit of atomicity, recovery, integrity and visibility by users. The trend in transaction support for design applications is to provide these capabilities at different levels of granularity. A designer involved in a design session might not want to make his work visible to other users. But would wants to take savepoints to avoid loosing work in a system crash.

A number of transaction models for processing of these advanced database applications have been proposed [Korth90], and some more abstract frameworks

[Ramamritham90] are being proposed. Some notable models and frameworks have been discussed below.

The nested transaction model distinguishes atomicity from visibility. A transaction can have any number of sub-transactions each of which executes atomically relative to other. However, no change is visible to other users until the top level transaction commits. The co-operative transaction model has been proposed for groups of designers working together on a task. All designers have mutual visibility for the work within the co-operative transaction, but no other users see changes until the co-operative transaction commits. Other models include the Recoverable Communicating Actions which can support arbitrary computation topologies, have been proposed in the context of distributed operating systems where interactions are more complex. Co-operative transactions and transaction groups have also been suggested for capturing interactions between new applications.

Another model proposed for concurrency control is often referred to as the check-in/check-out model. In this model, when a user wishes to make a change to an object, the user checks out a version of that object to his private workspace. This effectively sets a write-lock on that version of object disabling other users from attempting to make alteration to the version of that object. Once the desired changes have been made to the object version, the user may make the update permanent by checking in the version of object.

2.2.10 Query Language

A vital component of any data model is the query language which it provides for querying database objects. The query language must provide facilities for specifying

queries in an ad hoc fashion. The other desirable characteristics of the query language include declarative nature, thereby allowing relatively complex data structures to be interrogated using a simple query. Furthermore, the query language must also be efficient and provide scope for query optimisation. Finally, the query language must be application independent, enabling use on a wide variety of database applications.

A variety of approaches have been proposed in an attempt to achieve these objectives. These include the use of a language based on logic, extending an existing query language with object oriented capabilities, and development of a new query language from scratch.

For systems based on logic, queries resemble Prolog clauses with an action being performed if the pattern is matched against the database. Another approach that is quite prevalent is based on extending SQL with Object Oriented capabilities [Shipman90]. The Iris [Fishman90] prototype uses Object-SQL (OSQL) - extension of SQL. OSQL adds the concept of object identity and multivalued properties to SQL. Furthermore, the notion of a relation is replaced by types and functions.

Yet another approach that is quite prevalent in the ODBMS is extending a programming language with persistence. As a programming language includes facilities for data definition and manipulation, the main requirement for an object oriented query language is to extend these mechanisms to allow data types which persist across program executions. Different approaches have been adopted to incorporate persistence into a programming language.

Examples of programming languages extended with persistence include OPAL, ObjectStore and Ontos [Andrews90]. OPAL, the query language of GemStone prototype

queries in an ad hoc fashion. The other desirable characteristics of the query language include declarative nature, thereby allowing relatively complex data structures to be interrogated using a simple query. Furthermore, the query language must also be efficient and provide scope for query optimisation. Finally, the query language must be application independent, enabling use on a wide variety of database applications.

A variety of approaches have been proposed in an attempt to achieve these objectives. These include the use of a language based on logic, extending an existing query language with object oriented capabilities, and development of a new query language from scratch.

For systems based on logic, queries resemble Prolog clauses with an action being performed if the pattern is matched against the database. Another approach that is quite prevalent is based on extending SQL with Object Oriented capabilities [Shipman90]. The Iris [Fishman90] prototype uses Object-SQL (OSQL) - extension of SQL. OSQL adds the concept of object identity and multivalued properties to SQL. Furthermore, the notion of a relation is replaced by types and functions.

Yet another approach that is quite prevalent in the ODBMS is extending a programming language with persistence. As a programming language includes facilities for data definition and manipulation, the main requirement for an object oriented query language is to extend these mechanisms to allow data types which persist across program executions. Different approaches have been adopted to incorporate persistence into a programming language.

Examples of programming languages extended with persistence include OPAL, ObjectStore and Ontos [Andrews90]. OPAL, the query language of GemStone prototype

extends the Smalltalk programming environment with the necessary classes to support persistence. The query language of Ontos is based on C++ programming language extended with a persistence object store for C++ classes. Finally, another such system is ObjectStore [Orenstein92] has been derived by incorporating persistence in the programming language C++.

The main debate between different approaches that have been proposed for object query language centres on the relative seamlessness of integration between data definition, data manipulation language, and application programming language. Embedding a database query language in an application programming language has long been recognised as a potential source of problem. An impedance mismatch results at the boundary of two different data models: one supported by the database query language and the other by application programming language. The task of reconciling between two data models is left to the programmer. The argument cited by proponents of the extended programming language approach is that their approach avoids the mismatch by enforcing a single data model that of the extended programming language.

However, the proponents of an extended relation query language point that a seamless integration between two approaches precludes the possibility of a single standard object oriented query language. Generally, the typing system of programming languages are completely different. The advantage cited is that of a single universally accepted object oriented query language. Though, it has been recognised that SQL has a number of limitations, benefit can be derived from having a commonly understood and accepted database query language.

2.2.11 Version Support

The representation of multiple object versions is essential to many applications for an ODBMS. Consequently, most ODBMS address issues of representing, controlling and accessing multiple object versions. There are three distinct approaches for incorporating object versions in an ODBMS. The first approach considers versions as a fundamental facility and the Object Data Model (ODM) includes primitives to deal with them as a central component of the model. The Iris system is an examples of this approach. In this system, types or classes may have many versions identified by a version number. As a consequence, instances of a type may have many versions each of which is identified uniquely by a version number. Primitive operations in the language can reference object versions, create new object versions and so on.

In the second approach versions are not considered to be a part of the data model but instead, a version service is provided by a layer of software implemented directly on top of the kernel data model. This approach is implemented by the Encore system which implements version control and complex object services as a separate layer on top of its basic data model.

The third approach considers versions to be an application issue. Hence, rather than provide a model of version control in the database, applications must set up their own model of version control. The application must set up their own types for recording object version control, suited to their particular needs. This approach is characterised by the GemStone ODBMS which allows version control information to be recorded by defining appropriate classes in the GemStone class hierarchy.

Each approach mentioned above has advantages and disadvantages. there are advantages in applying the first approach, that of integrating version services with the data model. Particularly, the version service can be used as the basis of other database mechanisms, namely schema evolution and concurrency control.

2.3 Conclusions

In the first part of this chapter, we have presented three database architectures. With respect to architectures, there are two viewpoints to extensibility One with the approach of building a complete DBMS, and providing hooks for extensibility. This approach has been adopted by the Postgres ODBMS. Mechanisms are provided to accommodate complex objects, user defined data types, and procedures as data types. ODBMS based on extending a programming language with extensibility also fall in this category.

The other camp uses the approach of building a custom DBMS for an application domain. Database system generators and object managers may be classified in this category. These systems aid in the construction of a domain specific ODBMS. While both approaches have their weakness, no architecture for extensible database architectures has yet appeared.

We, in the second part of this chapter, have surveyed some of the desirable characteristics of an ODBS. For each feature, a number of prototypes that provide various alternatives was presented. The features of extensibility of the type system were also presented. Extensibility and flexibility of the type system was identified as a desirable characteristic of an ODBMS system. These characteristics have strong applicability for

advanced database applications, as the type system may need to be extended for a specific application domain.

Furthermore, a flexible structure of objects is also desired to capture the semantics of objects in the advanced application domain. Such applications desire the ability to extend the structure of application domain objects. In the next chapter, we present a data model for advanced database applications, which is aimed towards providing extensibility and flexibility for such applications.

Chapter 3

A Flexible Object Oriented Data Model

3.1 Overview

In the previous chapters we have reviewed the need of a flexible object oriented data model for advanced database applications. A description of database architectures in context of flexibility that they provide was presented. We also identified some of the desirable characteristics of an ODBMS.

A variety of object oriented data models with a varying degree of formalism have been proposed [Beeri90] [OMG92] [Jagdish89] and [Atkinson90]. None of the proposed models encompass all features of an ODBMS. This may be partially attributed to the fact that object oriented systems support a rich collection of sophisticated data modeling and manipulation concepts. Moreover, the terminology relating to object orientation has not yet fully matured. As a consequence, it is not uncommon for people [Participants89] to be referring to different capabilities when referring to ODB and data models.

Broadly speaking, object oriented data models may be categorized into structurally and behaviorally object oriented data models [Brown91]. The structural view includes structured complex objects, object identity, and some notion of is-a relationship and inheritance. A database is a collection of data items connected by various relationships. One may regard a database as undirected graph with various objects and their types as nodes of the graph with relationships between objects as links. Some object oriented data

models [Banerjee90] allow schema evolution. In such models, new classes may be added or changed in a class hierarchy.

The behavioral view of objects emphasizes on encapsulation of data of an object along with procedures/methods which act upon data and inheritance of data and methods. Abstract data types in object oriented languages (referred to as classes) encapsulate private data of the object with public procedures called as methods. The argument for encapsulation is one of simplifying the construction and maintenance of programs through modularization. An object may be seen as a black box that can be constructed and modified independently of the rest of the system as long as its interface is not changed.

A number of object oriented data models have been proposed - varying in the level of formalism. Notable of such efforts include the Object Management Group (OMG) [OMG92], which is aimed at the standardization of the use of object paradigm for construction of distributed applications. Applications may be constructed from plug compatible components, the interfaces of which are specified in the model. Another model that deserves attention is proposed by Beeri [Beeri90]. This model aims at developing a formal framework for object oriented database systems. Despite this activity, there is no commonly accepted object model, nor is it clear that such a model can be developed.

In this chapter we present a flexible object oriented data model for advanced database applications. Firstly, we argue for the need of another object data model. After some discussions about the notational conventions used in this chapter, the flexible object model is presented. The model proposed facilitates flexibility and extensibility. New data types may be added to the type system and the structure of an object may be altered. The

various aspects of the object model are presented and in the end we draw some conclusions.

3.2 Motivation

The current ODBMS are limited in terms of flexibility they provide. Typically, the flexibility provided is limited to the definition of new data types. Some research prototypes [Carey90], [Batory90] provide flexibility in that they allow users in the development of an ODBMS customized for a specific application domain. The Kala [Simmel91] basket mechanism presents another perspective to flexibility in that only primitive data storage mechanisms are provided. These primitives may be used as building blocks for a domain specific data storage mechanism.

The strategy for the incorporation of flexibility in an ODBMS adopted by these approaches, is aimed at the construction of data storage mechanisms for a specific application domain. The solutions proposed are not general, in their ability to provide a mechanism for flexibility independent of an application domain and database in use. The mechanisms that these prototypes allow are to a large extent oriented towards the development of a customized DBMS.

On the other hand, there has been increasing demands placed on the database community for flexibility in modeling advanced database applications. Not only it is desirable to have a flexible object model, the model must also permit extensibility. It is not uncommon in advanced database applications to need addition of new classes and object with heterogeneous data types associated with them.

Such an addition of new classes and objects is not uncommon in advanced application domains. For example, in an Office Information System (OIS), we may have multiple kinds of documents, each with a complex internal structure and behavior. We may need to incorporate a new type of document which is based on previously defined documents but differs in it's behavior or adds some more data members/behavior. In a VLSI component data base, we may need to add a new component which acts as a building block for complex circuits. Such additions are not necessarily lack in the design aspects of the application. These arise as a consequence of dynamic nature of the application domain. Mechanisms must be provided to cater for such additions.

Typically such changes would necessitate the recompilation of the entire data that has already been stored in the database to accommodate the changes. Orion [Banerjee90] describes a taxonomy of schema changes. The taxonomy covers changes to contents of a node in the inheritance lattice, changes to a method and changes to the edge of a node. However, changes to the structure of an individual object are not accounted for in the taxonomy proposed. We believe that such flexibility in the structure of individual objects is highly desirable in advanced database applications, which by their very nature are highly dynamic environments.

Despite the intense research and prototypes in the area of ODBMS, there has been no universally accepted data model. A number of commercial systems and research prototypes have been implemented or proposed recently. This is quite unlike the relational systems in the 1970, when a single formal model for the relational systems was proposed and followed by commercial prototypes. The current state of ODBS is characterized by intense research, concurrently with implementation. The situation may be attributed to the fact that ODBMS supports a rich collection of sophisticated data modeling and manipulation concepts. In this chapter we endeavor to present an object model for

advanced database applications such as CAD, CAM, CASE, complex information system and office information systems based on the object paradigm.

3.3 Notational Conventions

Notational conventions are very important for establishing a common ground upon which further higher level concepts may be discussed. These conventions become more important when dealing with the object paradigm. The authors in this field have varied interpretations of terminology. To establish some commonalty between these interpretations is necessarily an impossible task. We, in this section do not aim towards establishing a common framework of terminology for the object paradigm. The primary objective of such a notation is to define commonly used concepts to avoid misinterpretation. In this section we describe the notation that is used for the description of an object model. Diagrammatic representations used in this section are also presented to save repetition in the following sections.

An object represents an individual, identifiable item, unit, or entity either real or abstract with a well defined role in the problem domain. Furthermore, we may define an object to be characterized by it's state, identity and behavior. Each object has a unique existence independent of others. Each object maintains the state of the application domain entity that it is representing [Booch91].

No object exists in isolation. Rather, objects act upon each other. The behavior of an object refers to how an object acts and reacts, in terms of state changes and message passing with other objects. The behavior is represented as an external interface that is used by other objects when some operations are desired to be performed.

Whereas an object is a concrete entity that exists in time and space, a class is only a representation of the abstraction. In the proposed object model, a class may be seen as representing the commonalty between class instances. A class may be defined as a set of objects that share common structure and behavior. An object is simply an instance of a class.

The properties which may be associated with an object, maintain the state of the application domain entities. The property of an object may be classified in two categories: Fixed properties and Variable properties. The diagrammatic representation of the Fixed and the Variable properties are represented in the Figure 3.1 below.

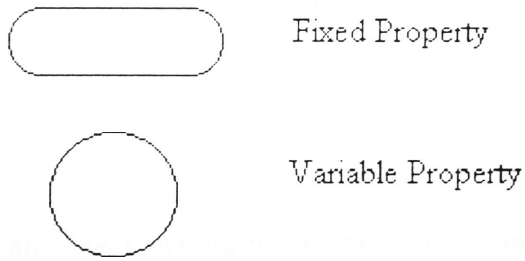

Figure 3.1. Schematic of Fixed and Variable Properties.

3.4 The Object Model

This section presents the flexible object model. Some concepts described herein have not been implemented in the prototype. The concepts in this section have been described informally. The implementation details and the functionality of the prototype are presented in Chapter 4.

One of the major objective in the development of a flexible object oriented data model was to extend existing object models in areas where they were found lacking. It

was not intended to develop a flexible object model from scratch. Such a model would require years of research and is not feasible.

The proposed model is characterized by it's simplicity, extensibility and flexibility. Despite the simplicity, the model is quite powerful in that it allows a high degree of flexibility in modeling advanced database applications. The model proposed deals with objects and their properties. Properties describe qualify or provide more information about the object with which they are associated. An object may have an arbitrary number of properties associated with it. The properties of an object may be broadly classified into fixed and variable properties.

3.5 Properties

The properties of an object maintain the state of an object. The state essentially records the structural characteristics of an object. The characteristics of the object need to be extended over time as an application evolves to cater to new requirements. As a consequence, new application domain entites need to be integrated with the application. This integration may include specialization of information that has been previously recorded or new information about application domain objects.

We define fixed properties of an object, the state which is conceptualized at the time of defining a class. On the contrary, the variable properties of an object represents extensions to the state of an object that have already been stored as fixed properties. The variable state information is applicable to individual instances of a class. An object may thus, be seen as constituting of fixed properties and variable properties. Figure 3.2 below illustrates a schematic of an object constituting of fixed and variable properties.

In the model proposed, properties can include data members of fundamental types as well as user defined types. Methods are generated for the retrieval of objects and their properties. It is possible to add new methods for classes in addition to those which have been defined. Addition of methods, however, expose the internals of the model. Consequently, the correct functioning of the model rests in the hand of the programmer defining them.

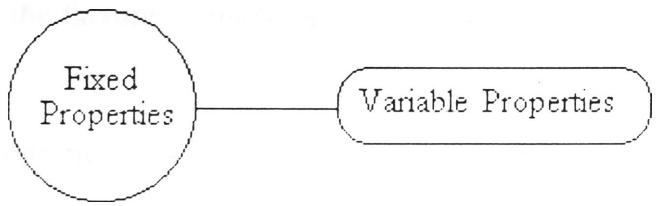

Figure 3.2. A Schematic of an Object

Properties may be associated with objects and may have an arbitrary complex internal structure. The variable properties of an object are implemented as independent classes which have been derived from a common ancestor "Property". Each property may have a complex internal structure, independent of the class to which it is associated. The members of a property may include the elementary data types, and other user defined data types. Thus, it is possible to define a property which has attributes of type complex numbers, imaginary numbers etc., as long as these types have been defined appropriately.

Such a mechanism provides flexibility to the application programmer as he is no longer limited to the data types which have been defined in the database. One is free to implement new data types, their relevant methods and use them as building blocks for applications. Such extensibility of the type system permits the programmer to define new data types based on the application domain.

3.5.1 Fixed Properties

As discussed in the previous section, the fixed properties of an object record the fixed state of the object. The fixed properties of an object are properties which have been defined at the time of creation of a class. Furthermore, the fixed properties of a class may be considered as attributes which are common to all instances of a class. Thus, fixed properties represent the factored commonality of class instances.

The fixed properties of a class are represented as data members of a class. These fixed properties are valid for the entire lifetime of the class, i.e. unlike variable properties, there is no provision for the deletion of fixed properties. These properties come into existence when a class is defined. The data types of the fixed properties is not limited to the built in data types. The user has the option to extend the type system by defining new application specific data types.

3.5.2 Variable Properties

Frequently, in advanced database applications we need to store property information specific to individual objects. Each object of a class may need to have a variable and heterogeneous set of properties associated with it. These properties typically have semantic relationships among each other. Such properties may represent specialization of some properties that have been stored with other objects in the class lattice. For example, in a VLSI design application, we may wish to maintain information about various components such as resistors, transistors, IC's etc.

The variable properties represent incremental additions to the state of an object. Such state information includes specialization of properties which may have already been

defined, or may represent new abstractions of the application domain. The new classes so formed either generalize or specialize from the pre-defined classes.

The variable properties that we wish to associate with a class, may not necessarily apply to all instances of the class for which a variable property has been defined. The state that we wish to record may only be applicable to a few instances of the class. On the contrary, the fixed properties maintain the state of an object that is applicable to all instances of a class.

The variable properties of a class are defined for an instance of a class rather than for all class instances. The variable properties of a class may be used to add information which is specific to class instances. This permits each instance of a class to have heterogeneous set of information associated with it. Such fine tailoring of properties which are applicable to an instance of a class is a unique feature of the flexible object model. Another consequence of variable properties is that the properties which are associated with a class instance may be added and deleted based on the application specific semantics.

3.6 The Core Class

The core class refers to the class which is being modeled from the problem domain into the application domain. Such a class typically represents the abstraction of the application domain. For example in a VLSI database, a component or circuit_type may be a useful core class. Each such core class is the root of a class hierarchy. Classes which are defined as being derived from the core class are grouped together in a class hierarchy rooted at the core class. Thus, all components or types of a circuit are derived from the core class.

3.7 Inheritance

Inheritance is a language feature that is used to express relationships between classes. Inheritance can be used in two different ways; compositional inheritance and a-kind-of inheritance. Compositional inheritance involves composing a new class from a class or classes that have properties in common with the new class. Class truck could be derived from an existing base class car, by deleting the property boot and adding the property load_type. [Myer92] A truck is not a kind of car, but has some properties in common.

A-kind-of inheritance is where the derived class really is a kind of its base class. The class truck is a kind of vehicle. In a-kind-of inheritance virtual properties become useful. A vehicle may have virtual properties number_of_doors and load_capacity.

In this section we describe inheritance of classes and properties. The classes which are defined in the model, along with their properties share the fixed and variable properties of their ancestors. This is useful in allowing new classes to be defined which are incremental refinements of a previously defined class. Inheritance allows to base a new class from an existing class.

3.7.1 Class Inheritance

In this section we discuss class inheritance. Class inheritance allows new classes to be derived from pre-defined classes. The classes which are defined in the database form an acyclic graph with the nodes of the graph formed by classes, which have been defined. The nodes in the database are linked together by various relationships. All classes in a hierarchy inherit directly or indirectly from the core or the root class.

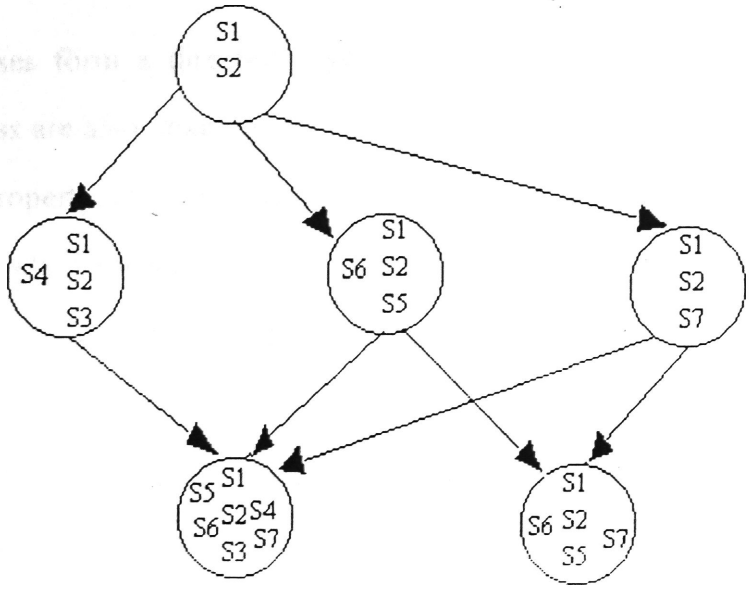

Figure 3.3. Schematic of a Class Hierarchy.

Figure 3.3 above illustrates a class hierarchy. In the schematic S[n] are the fixed properties. For the clarity of illustration of fixed properties, the variable properties have been omitted from the class hierarchy. The arrows linking the classes represent a-kind-of relationship. As illustrated, the root of the class hierarchy has two fixed properties labeled S1 and S2. These fixed properties are inherited by all classes which are derived from the core class.

3.7.2 Property Inheritance

A variable property inherits the state and behavior from other variable properties which have been defined. This is possible as all variable properties are internally represented as classes. All variable properties are derived from a global class "Property". Variable properties are defined for classes, however their instances are attached to class instances. The methods of properties are virtual and are refined in the derived properties. The rules relating to variable property inheritance have been described below.

The classes form a directed acyclic graph. The variable properties which are defined for a class are also linked in an acyclic graph. A distinct variable property graph, rooted at class property, exists for each core class. Thus, each core or the root class is the most appropriate place for the definition of variable properties which are applicable for all class instance and instances of classes which have been derived from the core class.

An aspect that deserves special merit relates to the semantics of association of variable properties with classes. Variable properties are defined for core classes or classes derived from the core class. However instances of variable properties are attached to class instances. A varible property instance may be attached to a class for which it has been used as a super class for any of the properties defined in the class lattice.

Figure 3.4 below illustrates the association of variable properties with classes. All variable properties inherit from class "Property". For example, the core class may have variable property P1 and P2 defined for it. Thus any instance of class S1 may have an instance of variable property P1 or P2 associated for it. Instances of a class may also define multiple instances of a variable property.

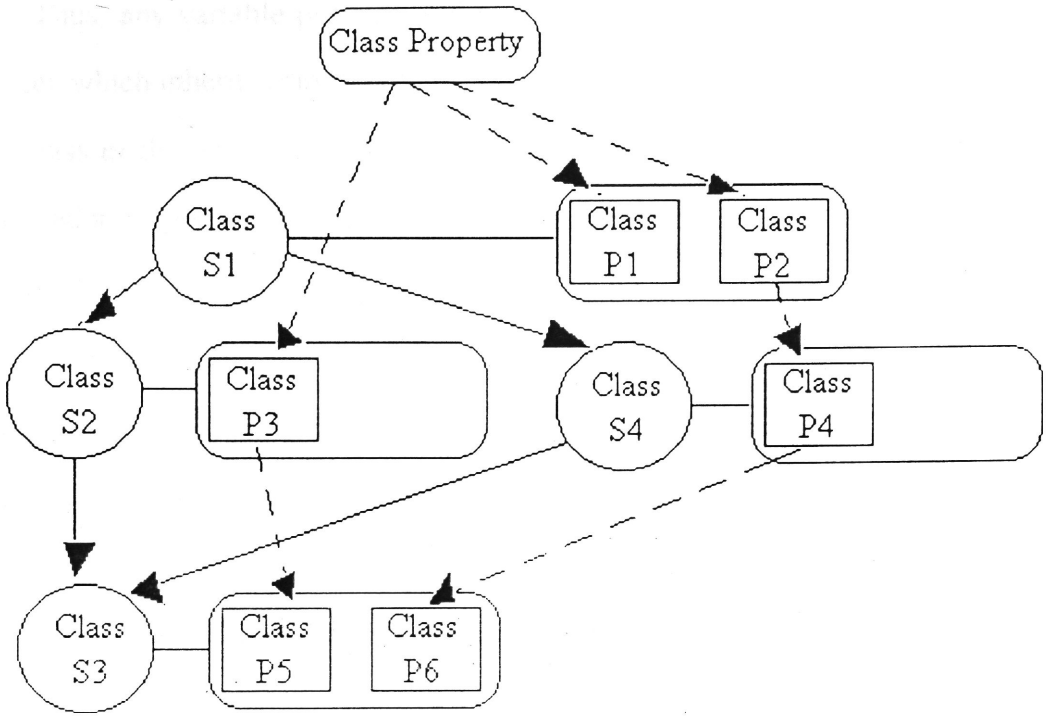

Figure 3.4. Property inheritance.

Variable properties may be declared for any class which is defined in the class hierarchy. Furthermore, a variable property defined for a class may inherit from other variable properties which have been defined. A variable property defined for a class may be used as an ancestor for any new variable property which is defined. For example property P2 inherits from the Global class property, and is also used as an ancestor for property P4.

However, a variable property may only be ancestor to variable properties of classes which lie under it in the inheritance graph. For example, in the schematic shown, properties P2, P4, P5 and P6 may inherit from the variable property P1 and/or P2 which has been defined for the root of the class hierarchy. As multiple inheritance is permitted, a variable property may inherit from multiple ancestors.

Thus, any variable property defined at the root of the class hierarchy is visible for all classes which inherit from the core class, both in terms of making instances and using as a base class in the inheritance lattice. For example, if we define a class named Employee and defined a property salary for it, then all classes which inherit from the class Employee automatically may associate instance of the property salary for instances of their classes. Any class which inherits from class Employee can have a property with salary as one of it's ancestors. This has the benefit in that variable properties which are declared at the top of the hierarchy can be defined for all classes which are specialisation's of the base class.

3.8 Addition and Deletion of Properties

Variable properties are defined for classes. Variable property instances however attach to class instances. In addition, variable properties may also be attached for classes which inherit from classes for which the variable property in question has been defined. Thus a variable property which is defined for a class hierarchy at the topmost level, can be used for all classes which inherit from the class.

Each class and variable property instance is identified by a unique identifier. This identifier is used for identification of class and variable property instances. The value for the identifier is assigned by the user and is used for the deletion of instances of a specific class and variable properties. When an instance of a class has been deleted, all variable property instances which have been defined for it are also deleted.

An important aspect of the proposed model is flexibility and extensibility. The prototype implementation allows the creation of new classes. Thus, new classes may be defined which inherit from pre-defined classes. Similarly, new variable properties may also

be defined dynamically, and instances made thereof. The instances of variable properties can be associated with class instances.

Such a mechanism permits extensibility and allows for the addition of variable properties and classes, each of which may be created from pre defined classes and added to the class hierarchy. Furthermore, the system is flexible in that it allows the dynamic addition of new classes and variable properties.

3.9 Constraints Checking

The object model allows the user to define new classes and variable properties which inherit from pre defined classes and variable properties respectively. As the definition of new classes and variable properties is under user control, there exists possibility of definition of new classes and variable properties which may result in inconsistent hierarchies. For example, if we have classes triangle, square and circle derived from abstract class shape and we wanted to define a new class polygon, then the classes listed in the inheritance tree as shape and triangle is not valid.

Figure 3.5 illustrates a schematic of conflict in the class hierarchy which can be detected with the constraint checking mechanisms built in the model.

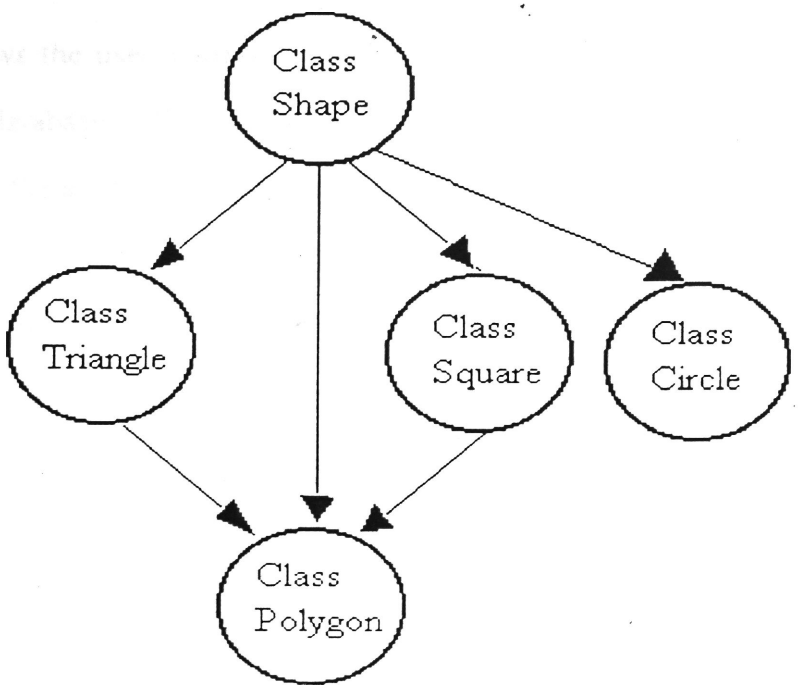

Figure 3.5. An Example of Class Hierarchy

Whenever a new class/variable property is defined that inherits from any other class/ variable property defined in the hierarchy, a check is made to ascertain whether it is valid to inherit from ancestors. The constraint checking is specific to checking consistencies of class and variable property hierarchies when a newly defined class or property is added.

3.10 Declarative Query Language

A declarative query language is proposed for the described model. The language places emphasis on ease of use and expressiveness The language allows the retrieval of class instances and their properties in an interactive fashion. It is proposed that such a query language would be beneficial for users and application domains that need to handle a large amount of data and users are not computer or database experts. The query

language allows the user a certain degree of flexibility in terms of being able to browse through the database. This approach is superior to the case wherein, the user must explicitly state the access path of the data to be retrieved.

The proposed scheme is based on concepts of sets. Sets are the unit of data that are retrieved from the database which satisfy certain criterion. Furthermore, the result of querying a class may be used for the set of objects on which selection criterion may be enforced. The objects in the set, at any instant represent those which satisfy the condition. Further queries are then applied to the collection of objects which had been retrieved in the previous case.

This scheme has the advantage of simplifying the normally tedious process of retrieving data from the database for browsing. Furthermore, objects obtained as a result of applying constraints may be reused to narrow down the further search of objects which satisfies additional constraints. Thus, the process of querying through the objects reduces to applying various constraints to the set of objects.

Variable properties which are associated with a class instance may be queried in a similar fashion. The object whose properties need to be queried is selected firstly. Having selected the object of interest, constraints may be applied upon the set of properties. However, only one class may be queried at a time.

3.11 Conclusions

In this chapter we proposed a flexible object oriented data model. The proposed model enabled flexibility in modelling advanced database applications. The flexible data

model for advanced database applications deals primarily with objects and their properties. Properties are associated with objects. The properties record the state of an object.

The properties of an object may be categorized into two broad areas. The fixed properties of the class are those which are defined at the time of creation of the class. Variable properties represent incremental additions to the state of an object. Objects may have heterogeneous properties associated with them. A user friendly query language based on concepts of sets is proposed for the model.

The model enables extensiblity and flexibility in modeling advanced applications. New data types specific to the application domain can be added. Furthermore, an object may have properties attached to it. These properties are individual classes and thus, may have an arbitrary structure. Application domain objects may have heterogeneous set of properties associated with them.

Implementation of flexibility in an existing ODBMS is an ambitious task. Such a task becomes even more difficult if the ODBMS does not provide any hooks for schema evolution as in the case of ObjectStore, the implementation platform. The next chapter discusses an approach to implementation of flexibility.

Chapter 4

Implementation

In this chapter, we present the design and implementation of the model that has been proposed in the previous chapter. Some sample class definitions and their methods have also been presented for the sake of illustration of concepts.

4.1 Overview

A prototype of the described model has been implemented under a commercially available object data base management system (ODBMS): ObjectStore under UNIX operating system. ObjectStore is a next generation ODBMS targeted at advanced database applications. ObjectStore falls in the category of persistent programming languages, being integrated with the C++ programming language. Support for persistence is orthogonal to type. Thus, objects of any type may be made persistent. Facilities are provided for concurrency, versions, and recovery.

The prototype inherits many features from the development environment (ObjectStore). Many features and functionality of the prototype have been based on ObjectStore features, or in cases where necessary by modifying or extending upon them. The prototype allows creation of arbitrary number of class hierarchies. Each class in the hierarchy may inherit from an arbitrary number of other classes. Furthermore, the inheritance tree may be arbitrary deep. New classes may be added to the class hierarchy and objects may have properties which are heterogeneous. The prototype also allows users to make new instances of a class, associate properties with a class, delete class and variable property instances and query classes.

4.2 Core Class

The core class may be seen as enveloping the root of a class hierarchy. For each hierarchy defined in the prototype, there exists a core class. Each core class contains a collection of pointers, which has the same type as that of the root of class hierarchy. The collection acts as a place holder for objects of root class and instances of all classes which have been derived from the base class.

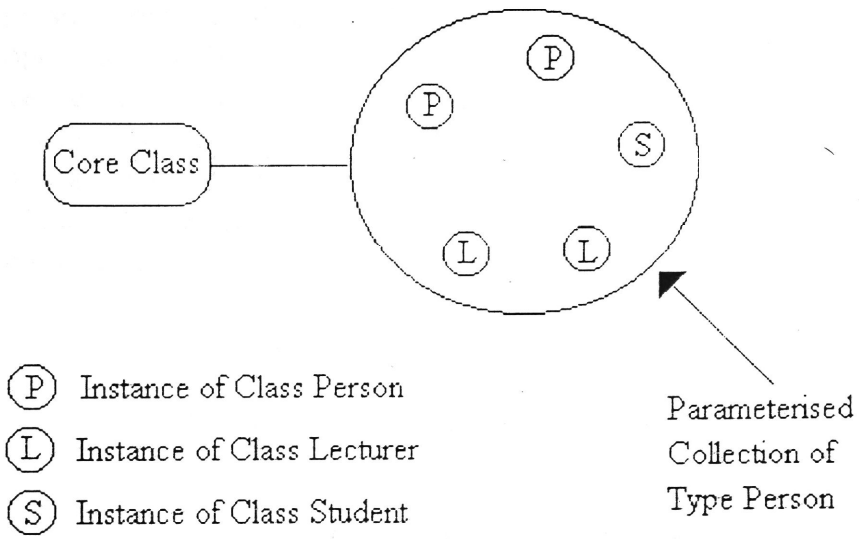

Figure 4.1 The Schematic of a Core Class

Figure 4.1 shows a class hierarchy which is rooted at class Person. The collection associated with core class contains all instance of class Person and all classes which have been derived from class Person. In the above figure, classes Lecturer and Student have been derived from the class Person and thus all their instances may be placed in the collection associated with the core class.

In response to the definition of a class corresponding to the root of a class hierarchy, class definitions, core class definitions, and associated methods are generated. The classes which together compose a class hierarchy include the core class, the class of the type defined, property class, and the property list class. A detailed description of these

classes appears later in this chapter. Figure 4.2 shows the class definition of the class person_core.

```
class person_core
{
        public:
        os_Collection<person*> person_list;
        person_core(): person_list(os_Collection<person*>::create(this)) { }
        ~person_core() { }
        os_Collection<person*> person_core::put_person();
        void person_core::initialise(char *path);
        void open_db(char *path);
        void delete_class_instance(char *c_id);
        void delete_property_instance(char *c_id,char *p_id);
        void display_person(char *c_id);
        void display_property(char *c_id);
        void insert_prop(char *c_id, property* a_property);
};
```

Figure 4.2 A Core Class Definition

The core class contains the public data member person_list which is a parameterized collection of type Person (root of the class hierarchy). This collection contains all instances of class Person and all classes which have been derived from class Person.

The initialise method is responsible for initialising the database, and the open_db method is responsible for opening the appropriate database for transactions. Methods are also included for deleting a class instance, deleting a property instance, displaying class and property instances. Methods for deleting a class or property instance takes as an argument the identifier of the instance to be deleted. Beside the initialise and the open_db method, the other methods associated with the core class invoke the appropriate methods depending upon the type of the object.

These class definitions and the associated methods are compiled and placed in a directory, identified by the name of the class which is at the root of the class hierarchy.

4.3 Classes

All classes which are defined in the class hierarchy may inherit from one or more pre-defined classes. Figure 4.3 below shows a graphical illustration of the class hierarchy. Consider a class hierarchy rooted at class Person. Subclasses of class Person, class Student and Lecturer may be defined. Furthermore, class RA, which inherits from class Lecturer and class Student may also be defined.

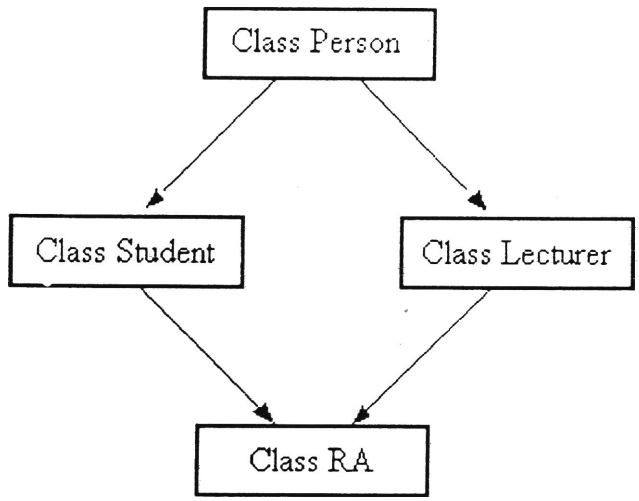

Figure 4.3. An Example of Class Hierarchy

Each class instance contains a data member which is a pointer to a class property_list. Property_list is a collection of type property. All variable properties defined for a class instance are placed in the property_list. Each instance of a variable property provides methods for making a new instance of itself, displaying and deleting itself. Thus, each variable property instance is a behaviourally complete object, in that it encapsulates data members and methods which operate on the data.

Each class which is generated by the prototype includes a data member which is used for identification of an instance. This identifier is distinct from the concept of object identity. Object identity, as already discussed in the previous chapters, is used for unique identification of an object. Additionally, it is a system generated identifier and is guarantied to be unique. The identifier which is generated with a class, however, is used for the identification of an object and it's value is assigned by the user.

```
class person
{
        property_list* person_prop_list;
        public:
        char*  _instance_id_;
        char*  person_name;
        int  person_age;
        person(char* _instance_id_f,char* person_name_f,int person_age_f)
        {
os_typespec *char_type = new os_typespec("char");
_instance_id_ = new(database::of(this), char_type, strlen(_instance_id_f)+1)
        char[strlen(_instance_id_f)+1];
strcpy(_instance_id_,_instance_id_f);
person_name = new(database::of(this), char_type, strlen(person_name_f)+1)
        char[strlen(person_name_f)+1];
strcpy(person_name, person_name_f);
person_age = person_age_f;
        }
        ~person() {}
        void make_new_person();
        virtual void display();
        virtual void display_prop();
        virtual void delete_prop_instance(char* p_id);
        virtual void insert_property(char *id, property *a_property);
};
```

Figure 4.4. The class

As shown in the Figure 4.4 above, the class Person contains the fixed properties: person_age and person_name. These properties are inherited by any class which is derived from class Person. Each class instance contains methods which are defined as

virtual. Virtual methods may be re-defined in a class which is derived from the base class. This allows us to specialise the behaviour of a class which is derived from an existent class. For the example above, in Figure 4.3 we can derive a class Lecturer from the class Person. Class Lecturer can have it's own methods for displaying, inserting, deleting etc. Additionally fixed properties which are associated with the class Lecturer may be defined which are in addition to those defined for class Person.

The method make_new_person calls the method to make a new instance of the class Person and adds it to the person_list collection. Since each class is distinct in how it's instances may be made, each class has a make_new method. After the instance has been created, it is inserted in the collection of instances associated with the core class. The display and display_prop methods have the task of displaying the fixed and the variable properties of a class instance.

The insert_property method takes as argument, an identifier of the object, and the property. The property is identified by a_property and is a pointer to an instance of property. This method inserts the property in the object. This method searches through the collection of instances which match the identifier. The property is then inserted in the list of properties associated with that instance (person_prop_list). The delete_property method takes as arguments, the identifier of the property to be deleted and searches for the list of properties associated with the instance and deletes the appropriate instance of property.

The virtual methods which have been defined in the classes have been defined as non-virtual methods in the core class. The core class invokes an appropriate method for each object. This is necessary as we would be adding new classes in the prototype without knowledge of their type. Thus in order to make new classes accessible in the

prototype, we must have a common ground on which they can be called. The core class is used for invoking appropriate methods of each object.

Each addition of a new class which is derived from a base class involves the generation of class definitions and associated methods of the new class. These methods and class definitions are compiled and linked with existing class definitions and methods of the class hierarchy.

The prototype of the flexible Object Oriented data model does not place any restrictions on the level and number of classes in the inheritance hierarchy. The user may define fixed properties of the class in addition to those which it inherits from.

In the prototype for a distinct class hierarchy many classes may be defined in an inheritance hierarchy. There also exists a distinct variable property tree for each class hierarchy. The metadata repository allows to store the definitions of classes which have already been defined. For subsequent invocations, as an aid to the user, tools are provided that allow to query the state of the classes and variable properties.

4.4 Properties

As described in the previous chapter, objects have properties associated with them. The properties which may be associated with a class are of two types: Fixed properties and variable properties. All properties of a class may not be known at the time of definition of a class. Furthermore the properties may be applicable to only a few application domain objects. Variable properties thus, make it possible for a user to add properties to a class, after the class has been defined and instances made thereof.

The class definitions of the property list class which acts as a place holder for all the properties which are defined in the model are shown in Figure 4.5.

```
class property_list
{
        public:
        os_Collection<property*> prop_list;
        property_list()
        {
          os_Collection<property*> prop_list = os_Collection::create(this);
        }
        ~property_list() { }
};
```

Figure 4.5. The Property List Class.

4.4.1 Fixed Properties

Fixed properties maintain the state of an object. They are defined at the time of definition of a class and remain valid for the entire lifetime of a class instance. The fixed properties may be thought of as attributes of a class. The fixed properties in the current implementation of prototype are implemented as data members of a class.

The data types of fixed properties include the elementary data types. New data type definitions may be incorporated in the prototype. These data types may be made use of in the definition of classes and properties.

4.4.2 Variable Properties

The variable properties of a class may be thought of as incremental additions to the state of an object. Such state information is typically added to an object after the state of an object has been conceptualised. Furthermore, the variable properties associated with a class are specified on a per instance basis rather than on class basis. Since making a new definition of the class would make the previous instances of the class inaccessible, variable

properties are implemented as separate classes. In response to a user desire of addition of variable properties to a class, a new class definition is generated. The generated class is derived from a global class property. This makes it possible for variable properties to be inserted in the collection associated with a class instance. Figure 4.6 illustrated the class definition and the associated methods of the global class property.

The class property has a single data member _prop_id_, which is used for identification of variable property instances. All properties which are defined in the prototype are derived from the class property. The only method associated with the global class property is the display method. This method displays data members of the property class. Each property has its own specific mechanisms for displaying its data members as data members associated with a variable property

```
class property
{
        public:
        char* _prop_id_;
        property(char* _prop_id__f)
        {
os_typespec *char_type = new os_typespec("char");
_prop_id_ = new(database::of(this), char_type , strlen(_prop_id__f)+1)
char[strlen(_prop_id__f)+1];
strcpy(_prop_id_,_prop_id__f);
        }
        ~property() {}
        virtual void display();
};
```

Figure 4.6. The Global Class Property

may differ. The display_method associated with a property is defined as virtual in the class definition of the Global property class. Each new property redefines the display method.

As in the case of addition of new classes to a class hierarchy, new variable properties defined for a class cause the generation of class definitions and methods. Each variable property which is added to a class instance defines methods for making a new instance, deleting itself corresponding to a given property identifier, and displaying its data members. The class definitions and associated methods are compiled and linked with the executable corresponding to the class hierarchy for which a property has been defined.

4.5 Inheritance

The inheritance hierarchy of classes and properties in the model is rooted at a single class which is class property for the case of properties and root class for classes. A problem arises in the multiple appearance of one class in the ancestry of another, multiply defined class [Coplien92]. This is commonly referred to as fork and join inheritance. For example, take the case of an abstract class window. Classes window_with_border and window_with_menu are derived from the abstract class window. These classes add to the functionality and the state of the abstract windows class. A problem now arises if we want to define a new class window_with_border_and_menu which inherits from window_with_menu and window_with_border, as we get two instances of class window. one from window_with_border and the other from window_with_menu.

This problem has been solved in the prototype by making some classes as virtual. Specifically, all classes which have been derived from a common ancestor are defined as virtual classes. This means that even if the base class appears several times in the derivation, it's data will appear only once in an instance of the derived classes. The class definition which are generated for classes and variable properties take into account the type of inheritance to be used.

4.6 Constraints Checking

As described in the above sections, the prototype allows classes and properties to inherit from pre-defined classes and properties. There exists a considerable scope for errors in that a user may define a class which inherits from other classes in an inconsistent manner. For example, consider a class hierarchy rooted at class Person and classes Student, Lecturer and Employee derived from it. A new class derived from classes Person and Employee would be inconsistent as class Employee already inherits from the class Person. The prototype incorporates constraint checking by detecting cases before generating the code for the class definitions and the corresponding methods.

4.7 Query Language

The prototype allows some primitive features of the query language. In this section we discuss the implementation of these features of the query language. This section describes the menu driven interface of the prototype. An alternate interface to the object data model has also been developed. This interface allows the user to programmatically perform the same functions that are typically carried out by the user in the interactive mode. A description of the interface is provided in the appendix.

The prototype implementation of the model permits the creation of class hierarchies, addition of classes to the class hierarchy, addition and deletion of variable properties of an object and querying. The methods of the prototype may be classified into the following categories.

1) Making a new class hierarchy
2) Adding a class to a class hierarchy
3) Adding a property to a property hierarchy
4) Checking class consistency

5) Checking property consistency

6) Making a class instance

7) Making a property instance

8) Deleting a class instance

9) Deleting a property instance

10) Querying

The main file associated with each executable, is responsible for invoking the various functional aspects of the prototype - depending upon arguments which are passed. Thus, to delete an instance of a class or property, the delete_instance method is invoked with arguments, class name, and _instance_id which uniquely identifies the class instance.

```
if(*argv[1] == 'a')
{
        core->open_db("/gkv/person/db1");
        myinterface->new_class_instance(argv[2]);
}
else if(*argv[1] == 'b')
{
        core->open_db("/gkv/person/db1");
        myinterface->new_property_instance(argv[2], argv[3], argv[4]);
}
else if(*argv[1] == 'c')
{....
```

Figure 4.7. Code Illustrating the Invocation of Appropriate Methods Depending upon Arguments Passed

The interface class may be seen as an interface between class definitions and the main file which is responsible for the invocation of the appropriate methods. As new classes are added to the class hierarchy, only the interface class needs to be rewritten to reflect the addition of new classes and properties. The main file checks for arguments which are passed onto it and calls the appropriate methods. Similarly, other features of

the query language have been implemented by invoking the executable file corresponding to the class hierarchy with appropriate arguments.

4.7.1 Defining Classes

Prior to making new instances of classes and their properties, classes and associated properties must already have been declared in the type system. A new core class or the class at the root hierarchy is defined by a menu driven interface in which types of various data members of the class are declared. A class which is derived from pre-defined classes in the class hierarchy is similarly defined.

In response to users desire of addition of a new class or property, the class definition and corresponding methods are generated and placed in a directory. To define a new class in the class hierarchy, it needs to be ascertained whether the new class is a base class or a derived class. If the new class to be added to the class hierarchy is a derived class, a check is made to ascertain if the new class is derived publicly from the base class or is derived virtually.

4.7.2 New Instances

Making new instance of a class is accomplished by identifying the class whose instance is desired to be made. To make an instance of a class or a property, the corresponding class or property must already have been defined in the class hierarchy. Similarly, to make a new instance of a property, there must exist an instance of a class to which the property is to be added.

Instances of classes and properties are made by a menu driven interface. In response to the generation of a new class instance, the executable file corresponding to the class hierarchy is executed with appropriate arguments. This results in execution of the

make_new method associated with the class whose instance is to made. The method prompts the user for values of attributes of the class or associated property. After making a new instance of class, the instance is inserted in the collection associated with the core class. A property instance is inserted in the list of property associated with a class instance. _instance_id and prop_id are data members which are automatically added to the list of attributes of a class and property respectively. These data members are used for the identification of class and property identifiers.

4.7.3 Deleting Instances

Deletion of a class instance is accomplished by a menu driven interface in the delete menu. To delete an instance of the class, the user must provide the _instance_id of the class that is to be deleted. To delete an instance of a property in addition to identifying the instance of the class, the instance of the property to be deleted also needs to be identified. This is necessary because there may be many properties associated with a class instance.

Instance of class and properties may be deleted by executing the executable corresponding to the class hierarchy to which the class whose instance is to be deleted belongs. The executable file is invoked with the identifier for invoking the delete_instance method. The arguments passed to the delete instance method include the _instance_id of the class instance to be deleted and additionally the class name. The method invoked searches through the list of class instances, and upon finding one that matches the _instance_id of the class to be deleted and deletes it.

4.8 Querying

The implementation of a prototype allows a primitive version of the query language. The user can select the class which is to be queried. Upon presenting the _instance_id of on object, all properties of class instance (fixed and variable) are presented. The user can also give a class name and, all instances of the class and their properties are presented.

To query the class and variable property instances, the executable file corresponding to the class which is to be queried is executed. The executable file is invoked with the identifier for invoking the query_class or the query_instance method. These methods search through the list of class instances, and on finding the class instance invokes the display method for instances and for all properties of the class instance.

4.9 Meta Data Repository

Information about classes and their properties are stored in an information repository in the database. This repository contains class definitions and property definitions of class hierarchies. The repository is useful in that new classes and properties may be defined. Definition of new classes may use the class definitions of the pre-defined classes which are stored in the repository. Furthermore, the class and property definition repository is useful in that the class definitions and the properties which have been defined previously are available to the user when defining a new class. Information about which properties have been associated with a class is also maintained in the meta-data repository. Such information is useful when a new class is to be defined which inherits from those which have been defined previously.

4.10 Schema Evolution

It was necessary to compile the definition of the class into an executable module, as the current version of ObjectStore does not permit schema evolution. The only method

of implementing objects with heterogeneous properties was to compile class definitions into an executable module. This module is executed with arguments for various functionalities.

Compiling class modules and executing modules to query a class is not a very efficient means. Besides, the compilation of modules take up a fair bit of space, in addition to the time for the compilation. In the current prototype, only the generated classes need to be compiled and linked with the executable module.

4.11 Conclusions

In this chapter, we have presented the implementation of the flexible object oriented data model. Each object is composed of fixed and variable properties. In the prototype of the proposed model, fixed properties of a class are implemented as data members of a class. Variable properties are implemented as individual classes which inherit from the global class property. The variable properties of an object are stored in the collection associated with the object. The various class definitions and methods which are generated by the prototype in response to the addition of a new class or property were also discussed. The primitive mechanisms that have been implemented for querying the database were also presented.

Chapter 5

Conclusions

In this thesis an attempt has been made to develop a flexible object oriented data model for advanced database applications. The advanced database applications are characterised with a need for flexibility in modelling advanced applications and extensibility to tailor to semantics of the application domain. Other characteristics of advanced database applications have been discussed in chapter 1.

5.1 The Flexible Object Model

One of the objectives of this thesis was to extend upon the work done in the area of object oriented data models. It was not intended to develop a data model from scratch. Consequently, additions to the object model have been proposed with a view to elevate some of problems in the area of advanced database applications. Two key features focused in this thesis, in the context of advanced application domain, relate to extensibility and flexibility. Suitable additions have been proposed.

The proposed model emphasises on the flexibility and extensibility. The primitives of the model include objects and their properties. Properties of an abject represent the state of an object. The properties of an object may be categorised into fixed and variable properties. The fixed properties represent the fixed state of an object. The variable properties of an object represent incremental additions to the state of an object. Such categorisation is a unique feature of the object model and permits to capture the semantics of the application domain.

5.2 Further Research

Further research should concentrate on the development of a formalism for the model proposed. A formal syntax and semantics of operations of the query language proposed in the chapter 3 are also suitable for formalism. It is also possible to improve upon the strategy for implementing flexibility in the prototype through features provided in version 2 of ObjectStore.

Appendix A

Implementation Guide

In this section, we present an implementation manual for the use of the flexible object model. This discussion is oriented towards the use of functionality of the data model by an application programmer. Firstly, an overview of ObjectStore ODBMS is presented. We, then, present a brief discussion of the object models proposed. This overview may necessarily be seen as repetition of concepts which have been described in chapter 3 and 4, but is provided here for the sake of completeness. Those with a desire to understand the theoretical concepts and obtain a complete picture, are referred to chapter 3 and 4 for further details.

A.1 The Environment

ObjectStore is a next generation ODBMS, aimed at advanced database applications. Support for persistence is orthogonal to type. Thus, objects of any types may be made persistent. ObjectStore falls in the category of persistent programming languages, being integrated with the C++ programming language. In this section we present an overview to ObjectStore [ODI91]. Issues related to persistence and schema information are discussed.

ObjectStore organises persistent data on disks. Each such disk contains one or more ObjectStore file systems, which contain ObjectStore databases. Each database is made up of segments, which are variable sized regions of memory that can be used as the unit of transfer from persistent storage to program memory. Each segment is made up of pages.

In addition to this physical organisation of data in persistent memory, ObjectStore organises databases into logical directories which form hierarchical structures in a manner similar to UNIX directories. However, ObjectStore directory hierarchies are independent of the UNIX directory hierarchies. They form a logical rather than physical organisation, as two databases in the same directories may be stored on different file systems.

With ObjectStore, data is mapped between database memory and program memory completely automatically and transparently to the user. ObjectStore detects a reference in a running program to persistent data, and automatically transfers the segment containing the referenced data across the network to the application's cache. Then the page containing the referenced data is mapped into virtual memory. Sometimes referenced data may already be present in the application cache. In such a case, all that is required is the virtual memory mapping or data may already have been mapped into the virtual memory. Once data has been mapped into the virtual memory, access to it is as fast as access to regular, transient data.

ObjectStore stores schema information in each database. Schema information stored includes information about classes of objects stored there and layout of instances of those classes. This allows ObjectStore to identify boundaries of pointer fields in each newly retrieved segment. Such information can be used to ascertain if the referenced database memory is currently mapped into virtual memory or must be assigned to unused virtual memory.

Schema information is stored as C++ objects. As classes are not run-time objects in C++, so ObjectStore must generate representation of classes in order to manage database memory. These representations are generated at compile time - for each application that might store information in a database. So, at run-time, when the application stores an object in a database, a representation of the object class is ready to be

added to the database schema along with the object itself. If instances of that class already exist in the database, the application's class representation is checked against that of the database to make sure that they agree. Information about application's schema (generated at compile time) is stored, not in a UNIX file as with object code and the executable, but rather in an ObjectStore database.

The generation of an ObjectStore schema consists of two processes. The first step in the generation of schema information takes place during compilation, and the second step occurs during linking. In the first stage, a preliminary schema is created, referred to as the compilation schema database. As each source file is compiled, the use of each class in the file is noted. If a class is one that serves as entry point for retrieval from the persistent storage or one whose instances are persistently allocated, an object representing it is stored in the compilation schema database. In addition, for each class reachable by navigation from one of these classes, an object is stored in the compilation schema database as well. In the second stage, during the link step, all objects in the compilation schema database are brought together with objects representing any library schemas used in the application. These objects are stored in the application schema database, the ultimately repository of information on an application's schema.

A.2 The Flexible Object Model

The object model proposed, is aimed at providing flexibility in advanced database applications. The model deals primarily with objects and their properties. The properties of an object may be categorised into fixed and variable properties. The fixed properties of an object record it's fixed state. The variable properties of an object represent incremental addition to the state of an object. An object may have an arbitrary number of heterogeneous properties associated with it.

The class and properties form two distinct hierarchies. The classes are arranged in an acyclic graph which is rooted at the core of the class hierarchy. Similarly, properties are arranged in an acyclic graph which is rooted at class Property. All properties inherit directly or indirectly from the global property class.

For the purpose of identifying the class and property instances, each class or property instance is assigned a unique identifier. This value for this identifier must be given by the user. This identifier must be used when retrieving properties of an object or when adding deleting a property instance.

A.3 The Methods

In this section we present a discussion of the functionality of the object model. The prototype implementation of the model permits the creation of class hierarchies, addition of classes to the class hierarchy, adding and deleting properties to an object and querying the classes. The methods of the prototype may be classified into the following categories.

1) Making a new class hierarchy
2) Adding a class to a class hierarchy
3) Adding a property to a property hierarchy
4) Checking class consistency
5) Checking property consistency
6) Making a class instance
7) Making a property instance
8) Deleting a class instance
9) Deleting a property instance
10) Querying

The prototype of the implemented model is oriented towards an interactive user. In the current prototype, the user may define new classes, new class hierarchies, and associate new properties to objects of a class. The user is presented with a screen oriented menu which allows the user to select the operation to be performed.

An alternate interface to the object data model has also been developed. This interface allows the user to programmatically perform the same functions that are typically carried out by the user in the interactive mode. The definition of all methods which are described in this appendix are contained in the file prog.hh. All programs that utilise methods which have been indicated in this appendix must include the file prog.hh.

A.3.1 Making a Class Hierarchy

The purpose of this method is the creation of a new class hierarchy. The class which is at the root of the class hierarchy is the ancestor of all classes defined in the class hierarchy. A new class hierarchy is created by invoking the following method.

make_new_hierarchy (char* core_name, char* path)

Where core_name indicates the name of the class which is at the root of the class hierarchy. Path indicates the full path name of the directory wherein the class definitions, methods and the object modules are stored. Each class hierarchy defined in the system has a location in which class definitions and associated methods are stored.

After a class hierarchy has been created (not quite yet !), we must name its data members. The names of data members, their types along with information regarding values also needs to be provided. This method may be called repeatedly until the desired number of data members have been defined.

class_attr_name (char* a_name, char* a_type, char* val)

The argument a_name refers to the name of the attribute, a_type refers to the data type of the attribute. Val is an argument indicating the length of a character string. For other data types, set the value of val equal to 0.

In the prototype, only primitive data types are allowed. The types that may be used in the prototype include integer, long integers, floats, doubles and characters. There is no limitation on the length of characters that may be used in a string.

After defining all data members of a class, set a_name to _END_ to indicate the end of data members to be defined. This signifies that the class hierarchy needs to be generated. The class definitions and appropriate methods for displaying data members and primitive means of querying the database are also provided.

A.3.2 Adding a Class to a Class Hierarchy

A new class may be added to the class hierarchy by indicating the class from which it inherits. Furthermore, attributes/fixed properties must be specified. The name of the fixed property along with the type of attributes must also be indicated. To add a new class to an existing class hierarchy, the following method may be used.

add_class (char* class_name)

Where class_name is the name of the root class in the hierarchy. The root class on which a class hierarchy is based must have been previously defined before adding a new class.

Each class in a class hierarchy may have multiple ancestors. The data members of ancestors are inherited by subclasses. To specify the ancestors of a class, the following method may be called.

add_class_ancestor (char* ancestor)

Any number of classes may be named as ancestors of a class. However, ancestors defined for a class must have been already defined in the same class hierarchy. The argument ancestor is the name of the class which is the ancestor of a newly defined class. Each class defined in the hierarchy must have at least one ancestor to link it in the class hierarchy. Setting the value of ancestor to _END_ signifies end of the list of ancestors of a class.

class_attr_name (char* a_name, char* a_type, char* val)

As mentioned above, data members and their type must be defined. In the argument list a_name is the name of a data member of a class, a_type is the data type of the data member. Only elementary data types may be used as attributes in the class definition. The val argument is used to specify the length of the character field. There is no restriction on the length of the character field that may be used as a data member of a class.

A.3.3 Defining a New Property

In the flexible object model, properties are associated with objects. The properties of an object record the state of an object. Fixed properties are implemented as data members of a class, while variable properties are implemented as individual classes. Each

object of a class has a collection of properties associated with it. Creation of variable properties results in the creation of a new class.

A new property must be declared for a class before instances of that property can be associated with a class instance. Declaration of properties and making new instances of properties are different concepts. To declare a property for a class, the name of the property, it's constitute data members, and their types must be defined. The class to which a new property is to be attached must previously have been defined. A new property may be defined by the following method.

add_property (char* class_name, char* prop_name)

Class_name indicates the name of the class for which a new property has been defined and prop_name indicates the name of the new property.

As classes are linked together by inheritance forming a class hierarchy, properties have a distinct hierarchy rooted at a class named property. All variable properties inherit from the class property. Each variable property that is defined for a class may have multiple ancestors. The ancestors of a variable property must have been previously defined. Each variable property defined must inherit from at least one other variable property defined in the property hierarchy. The name of classes which are ancestors of the variable property are specified by the following method.

add_prop_ancestor (char* name)

The add_prop_ancestor method takes as parameter the name of a class. There is no limitation on the number of ancestors of a class.

The data members of the variable property may be defined by calling the prop_attr_name method. The argument of this method have the same semantics as that of the class_attr_name. Conditions relating to data types are similar to that of the class_attr_name.

prop_attr_name (char* a_name, char* a_type, char* val)

Where a_name is the name of the attribute, a_type is the data type of the attribute and val indicates the value of character string. Val should be set to 0, for other data types.

A.3.4 Checking Class Consistency

To check the consistency of the class which has been defined in the class hierarchy, the check_class_consistency method is provided. This method is responsible for checking the consistency of a class which has been defined in the class hierarchy. An integer value of -1 indicates that the indicated class is not consistency. The definition of the property defined is also removed from the meta-data repository. A value of 0 indicates that the class definition is consistent and has been added to the property hierarchy successfully.

int check_class_consistency (char* root, char* class_name)

Where, root indicates the root of the class hierarchy, and class_name is the name of the newly added class whose consistency is desired to be ascertained.

A.3.5 Checking Property Consistency

This method checks the consistency of the variable property which have been defined for a class. The consistency of variable properties is ascertained by the following method.

int check_property_consistency (char* class_name, char* prop_name)

An integer value of -1 indicates that the variable property indicated is not consistency. The definition of the variable property defined is also removed from the meta-data repository. A value of 0 indicates that the variable property definition is consistent and has been added to the property hierarchy successfully.

A.3.6 Making a New Instance of Class

To make an instance of a class, the class hierarchy to which the class belongs must be given as argument. The name of the class whose instance is desired to be made must also be identified. Furthermore, values of data members, including (_instance_id_) of the class must also be indicated.

new_class_instance (char* root, char* class_name,<values>)

Root indicates the root of the class hierarchy. Class_name indicates the name of the class whose instance is to be made. <values>represent values of data members of the class. The value of class identifier must be the first value in the list of values. It is assumed that names of classes are distinct in a class hierarchy.

A.3.7 Making a New Instance of Property

To make a new instance of a variable property, we must indicate the name of the class to which the variable property is to be added and the name of the variable property.

new_prop_instance (char* class_name, char* class_id, char* prop_name, <values>)

Class_name indicates the name of the class whose instance is to be made. Prop_name is the name of the variable property. _Instance_id is the identifier of the class to which the instance of the variable property must be attached. <values> represent values of data members of the variable property. Property identifier must be the first value in the list of values of the variable property.

A.3.8 Delete a Class Instance

Class instances may be deleted by the delete method. To delete a class instance, the identifier corresponding to the class instance must be given as an argument. The class identifier of a class instance identifies each instance of a class uniquely. This identifier is distinct from the object identity, in that it is used only for the identification of class instances. This method may be called as

delete_class_instance (char* class_name, char* _instance_id)

Where, class_name is the name of the class whose instance is desired to be deleted. _Instance_id is the unique identifier that has been associated with the class instance. In the current prototype, the programmer has the task of maintaining the class identifiers which are created. After the invocation of the delete_class_instance method, which takes as argument, the _instance_id of the object desired to be deleted. Any variable properties associated with the class instance are also deleted.

A.3.9 Delete a Property Instance

To delete an instance of a variable property associated with a class instance, the delete_property_instance method must be invoked. The delete property instance method searches through the list of variable properties associated with the class instance and deletes the specified variable property instance. The delete property instance takes as arguments, the identifier of the class as well as the identifier of the variable property that is desired to be deleted.

delete_property_instance(char* class_name, char* _instance_id, char* prop_name, char* prop_id)

Where class name is the name of the class to which the variable property has been defined. Class identifier indicates the identifier of the class instance, to which the variable property has been at attached. Property identifier is the identifier of the variable property which is desired to be deleted.

The delete property instance may be invoked by calling the delete_property_instance method which may be invoked in the following fashion.

A.3.10 Querying

To query a class hierarchy, the following method is provided.

os_collection<property*> all_props (char* root, char* _instance_id)

This method takes as argument the identifier of a class whose properties are desired to be queried. This methods returns a collection of type property which contains

all properties of the identified class instance. The programmer may then use these instances in an appropriate manner in an application program.

References

[Afsarmanesh90] Afsarmanesh, D., et al., "An Extensible Object-Oriented approach to Databases for VLSI/CAD," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 607-618.

[Agarwal89] Agarwal, R., Gehani, N.H.,"Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++," Hull, R., Morrison, R., Stemple, D.(Ed.), "Database Programming Language," Proceedings Of The Second International Workshop on Database Programming Languages, California: Morgan Kaufmann Publishers Inc., June 1989, pp 25-40.

[Ahsen90] Ahsen, M., et al., "An Architecture for Object Management in OIS," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc.,1990, pp 570-581.

[Albano89a] Albano, A., et al.,"A Framework for Comparing Type Systems for Database Programming Languages," Hull, R., Morrison, R., Stemple, D.(Ed.), ."Database Programming Language," Proceedings Of The Second International Workshop on Database Programming Languages, California: Morgan Kaufmann Publishers Inc., June 1989, pp 170-178.

[Albano89b] Albano, A.,et al.,"Types for Databases: The Galileo Experiment," Hull, R., Morrison, R., Stemple, D.(Ed.), "Database Programming Language," Proceedings Of The Second International Workshop on Database Programming Languages, California: Morgan Kaufmann Publishers Inc., June 1989, pp 196-206.

[Andrews90] Andrews, T. and Harris, C.,"Combining Language and Database Advances in an Object-Oriented Development Environment," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990.

[Arango91]. Arango, G.,"O2," Communications of the ACM, Vol. 34, No 10, October 1991, pp 35-48.

[Atkinson90] Atkinson, M., et al., "The Object Oriented Database System Manifesto," Kim, W., Nicolas, J.M., Nishio, S., "Deductive and Object Oriented Databases," North-Holland: Elsevier Science Publishers B.V., 1990, pp. 223-240.

[Banerjee90] Banerjee, J., et al., "Data Model Issues for Object Oriented Applications," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc. , 1990, pp 197-208.

[Batory90] Batory, D.S., et al., "GENESIS: An Extensible Database Management System," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 500-518.

[Beeri90] Beeri, C., "Formal Models for Object Oriented Databases," Kim, W., Nicolas, J.M., Nishio, S., "Deductive and Object Oriented Databases," North-Holland: Elsevier Science Publishers B.V., 1990.

[Booch91] Booch, G., Object Oriented Design With Applications, California: The Benjamin/Cummins Publishing Company, Inc., 1991.

[Brolio89] Brolio, J., et al., "ISR: A Database for Symbolic Processing in Computer Vision," IEEE Computer, Vol. 22, No. 12, December 1989, pp 22-30.

[Brown91] Brown, A., Object Oriented Databases, McGraw Hill, 1991.

[Butterworth91] Butterworth, P., et al., "GemStone OODBMS," Communications of the ACM, Vol. 34, No 10, October 1991, pp. 65-77.

[Carey90] Carey, M.J., et al., "The EXODUS Extensible DBMS Project: An Overview," Zodnik, B. S, Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 474-499.

[Cattell91] Cattell, R.G.G., Object Data Management: Object-Oriented and Extended Relational Database Systems, Massachusetts: Addison-Wesley, 1991.

[Chrysanthis90] Chrysanthis, P.K., Ramamritham, K., "ACTA: A Framework for specifying and reasoning about Transaction structure and Behaviour," New York: ACM press, Sigmod Record, 1990, pp 194-203.

[Cockshott90] Cockshott, P.W., et al., "Persistent Object Management," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 251-272, .

[Coplien92] Coplien, O.J., Advanced C++ Programming Styles and Idioms, Massachusetts: Addison-Wesley, 1992.

[Date90] Date, C.J., An introduction to Database Systems, 5th ed., Massachusetts: Addison-Wesley, 1991.

[Dayal90] Dayal, U., et al., " Simplifying Complex Objects: The PROBE Approach to Modelling and Querying them," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 390-399.

[Dixon89] Dixon, G.N., et al., "The Treatment of Persistent Objects in Arjuna," Cook,S.(Ed.), "ECOOP 89: Proceedings of the Third European Conference on Object Oriented Programming, Cambridge: Cambridge University Press, July 1989, pp 169-190.

[Fishman90] Fishman, D.H., et al., "Iris: An Object Oriented Database Management System," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 216-226.

[Gehani92] Gehani, N.H., et al., "Event Specification in an Active Object-Oriented Database," Stonebraker, M., (Ed.), "Proceedings of the 1992 ACM Sigmod International Conference on Management Of Data", California: Sigmod Record, Vol. 21, No. 2, June 1992, pp. 81-90.

[Goldberg89] Goldberg. A., Robson D., Smalltalk-80: the language, Massachusetts: Adison-Wesley, 1989.

[Gupta91] Gupta, R., et al., " The Development of a Framework for VLSI CAD", Gupta R., Horowitz E., (Ed.) "Object Oriented Databases with Applications to CASE, Networks, and VLSI CAD", pp 237-260, 1991.

[Jagdish89] Jagdish, H.V. and Lawrence, O.,"An Object Model for Image Recognition," IEEE Computer, Vol.22, No. 12, December 1989, pp 33-41.

[Kent91] Kent, W.,"The Breakdown of the Information Model in Multi-Database Systems," New York: ACM press, Sigmod Record, Vol. 20, No 4, December 1991, pp 10-15.

[Khoshafian90a] Khoshifian, S., et al.,"Storage Management For Persistent Complex Objects," Information Systems, Vol. 15, No 3, January 1990, pp 303-320.

[Khoshafian90b] Khoshafian, S., Copeland, G.P, "Object Identity," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 37-46.

[Korth90] Korth, F.H., et al.,"On Long Duration CAD Transactions," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 408-431.

[Kotteman91] Kotteman, E. J., et al., "A storage and Access Manager for Ill-Structured Data", Communications of the ACM, Vol. 34, No. 8, August 1991, pp 94-103.

[Lohman91] Lohman, M.G.,"Extensions to Starbust: Objects, Types, Functions and Rules," Communications Of The ACM, Vol. 34, No. 10, October 1991, pp 94-109.

[Lamb91] Lamb, W. C., et al., "ObjectStore - An Object Oriented Database System, Communications of the ACM, Vol. 34, No. 10, October 1991, pp 51-63.

[Lochovsky90] Lochovsky, F.H., et al., "OTM: Specifying Office Tasks," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 582-591.

[Maier 89] Maier, D., Making Databases fast enough for CAD Applications, Object-Oriented Concepts, Databases and Applications, Addison-Wesley, 1989.

[Manola90] Manola, F. and Dayal, U.,"PDM: An Object-Oriented Data Model," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 209-215.

[ODI91] Object Design, Inc., ObjectStore Reference Manual: Release 1.1, For UNIX-Based Systems, Burlington,Massachusetts: Object Design, Inc., March 1991.

[OMG92] Object Management Group (OMG), "OMG Architecture Guide Chapter 4: The OMG Object Model,□ Object Model Task Force, May 1992.

[Orenstein92] Orenstein, J., et al.,"Query Processing in the ObjectStore Database System," Stonebraker, M., (Ed.), "Proceedings of the 1992 ACM Sigmod International Conference on Management Of Data", California: Sigmod Record, Vol. 21, No. 2, June 1992, pp. 403-412.

[Participants89] The Laguna Beach Participants, "Future Directions in DBMS Research," New York: ACM press, Sigmod Record, Vo. 18, No.1, March 1989, pp 17-26.

[Premeriani90]Premeriani, W.J., et al., "An Object Oriented Relational Database," Communications Of The ACM, Vol. 33, No. 11, November 1990, pp 99-109.

[Rowe90] Rowe, L.A., Stonebraker, M.R., "The POSTGRES Data Model," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 461-473.

[Shipman90] Shipman, D.,"The Functional Data Model and the Data Language Daplex," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 95-111.

[Schuh] Schuh, D., et al., "Persistence in E Revisited - Implementation Experiences," Wisconsin: University Of Wisconsin, Technical Reports.

[Simmel91] Simmel, S.S, Godard, I., "The Kala Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations," OOPSLA '91 Conference Proceedings, New York: ACM press, October 1991, pp 230-246.

[Snyder90] Snyder, A.,"Encapsulation and Inheritance in Object Oriented Programming languages," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 84-91.

[Stonebraker91] Stonebraker, M. and Kemnitz, G.,"The Postgres Next Generation Database Management System," Communications Of The ACM, Vol. 34, No. 10, October 1991, pp 78-93.

[Stroustrup91]Stroustrup, B., The C++ Programming Language, 2nd ed., Massachusetts: Addision-Wesley, 1991.

[Woelk90] Woelk, D., et al., "An Object Oriented Approach to multimedia Databases," Zodnik, B. S., Maier, D. (Ed.), "Readings in Object-Oriented Database Systems," California: Morgan Kaufmann Publishers Inc., 1990, pp 592-606.

[Zdonik87] Zdonik, S.B., Hornick, M., "A Shared, Segmented Memory System for an Object-Oriented Database," ACM Transactions on Office Information System, Volume 5, No 1, January 87.